The Dissertation Committee for Soon Hyeok Choi
certifies that this is the approved version of the following dissertation:

A Software Architecture for Cross-Layer Wireless Networks

Committee:

Scott M. Nettles, Supervisor

Dewayne E. Perry

Christine Julien

Sriram Vishwanath

Lili Qiu

A Software Architecture for Cross-Layer Wireless Networks

by

Soon Hyeok Choi, B.S.E., M.S.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2008

To my family

for their love and support

Acknowledgments

First of all, I would like to thank my supervisor, Dr. Scott Nettles, for his guidance and support. He guided me throughout my research at the University of Texas at Austin with remarkable insights and profound knowledge, and also encouraged me by showing me a great role model as a researcher, a teacher and also a human being.

I am indebted to Dr. Dewayne Perry for all his supportive guidance and advice on my research, to Dr. Christine Julien for all her advice and support, to Dr. Sriram Vishwanath for his kind encouragement and support, and to Dr. Lily Qiu for her helpful direction.

I would also like to thank colleagues of my research group, Seong-Kyu Song, Minyoung Park, Yihong Zhou, Sangwoo Lee, Gibeom Kim, Ketan Mandke, Wonsoo Kim, Robert Grant and Hyrum Wright for their helpful discussions and enjoyable working with them.

I would also like to thank my friends I met here in Austin, Byungchul Jang, Jinkyu Lee, Hongjoong Shin, Jisun Park, Hyunsoo Park, Kyungtae Han, Chulhan Lee and Eunho Choi for their friendship. My years in Austin were pleasant and enjoyable because of them.

Finally, I would like to dedicate this dissertation to my wife Joo-Young Oh, who has shared life and always supported me with love and encouragement, to my two daughters Minsun and Minyoung, who are always giving me invaluable happiness, and to my parents and parents-in-law for their love, support, and encourage-

ment. Without them, it would have been impossible to accomplish this.

Soon Hyeok Choi

The University of Texas at Austin

May 2008

A Software Architecture for Cross-Layer Wireless Networks

Publication No. _____

Soon Hyeok Choi, Ph.D.
The University of Texas at Austin, 2008

Supervisor: Scott M. Nettles

Conventional data networks are based on a layered architecture, in which a layer implements some aspect of the network while hiding the detailed implementation from the other layers. The introduction of wireless networks has created a need to violate this layered discipline to create cross-layer designs or adaptations. Such cross-layer adaptations optimize the performance of wireless networks by using information from any layer in the network. The key problem is that ad-hoc implementations of cross-layer adaptations introduce complex interactions between layers and thus reduce the level of modularity and abstraction in the network's implementation. This gives rise to a significant increase in complexity.

We demonstrate that a new software architecture is able to provide a systematic framework that helps us to implement a wide variety of cross-layer adaptations while preserving to a significant degree the modularity found in the existing network's implementation. To develop such an architecture, we first create a taxonomy of possible cross-layer adaptations. The taxonomy allows a precise description of a wide variety of cross-layer adaptations. Thus our taxonomy can serve as a framework for developing a cross-layer architecture.

We develop the software architecture by creating two architectures, a conceptual one and a concrete one. We first develop a conceptual architecture, which shows the key mechanisms that are required to implement cross-layer adaptations. This architecture helps us to understand how we can implement cross-layer adaptations by using our architectural framework. We then develop a concrete architecture, which shows how we can implement such a conceptual architecture on real wireless systems. This architecture addresses more detailed implementation issues. We design the concrete architecture for Hydra, which is a flexible wireless network testbed. We then show that our architecture is generic enough to allow us to support a wide set of cross-layer architectures.

We evaluate the proposed architecture by performing three case studies, each of which implements a cross-layer adaptation within Hydra based on the concrete architecture. The case studies allow us to implement and evaluate the key mechanisms provided by our architectural framework. We also implement each cross-layer adaptation by using a conventional approach, in which one layer performs the cross-layer adaptation directly communicating with other layers and other nodes. Comparing both the implementation techniques allows us to evaluate how our architectural framework supports a wide variety of cross-layer adaptations while reducing the complexity of implementation of cross-layer adaptations.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The success of the IP-based Internet can hardly be overstated. An important underlying key to the Internet's success is that its design and implementation are based firmly on a well established architecture, commonly referred to as the "hourglass model" [1]. As shown in Fig. 1.1, the hourglass model defines a set of layers, each of which implements some aspect of the network, while hiding the detailed implementation and complexity of that layer from the other layers. Thus the hourglass model allows us to freely change operations of a layer without significant impact on the other layers.

For networks that are composed of wired links such as most of the Internet, this layered architecture is remarkably successful. However, the introduction of wireless links has revealed the limitations of this layered architecture. The properties of wireless links such as high error rate, low and variable bandwidth, and long delay significantly impact the overall performance of wireless networks. Thus wireless networks need to optimize the performance of the network by using information from many layers of the network. This idea of "cross-layer" protocol design or adaptations has become a very active research area [2, 3, 4, 5], and recent research has shown that this approach can solve a variety of problems in wireless networks.

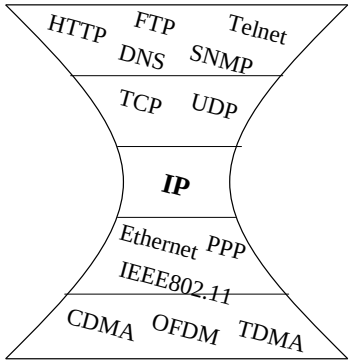


Figure 1.1: The hourglass model

The problem with cross-layer adaptations lies in violating the fundamental structure of the conventional layered architecture. Implementation of cross-layer adaptations introduces complex interdependencies between layers and substantial changes to existing protocol implementations, many of which are tailored to particular adaptation processes. These ad-hoc implementations compromise the modularity of existing protocol implementations and thus make it hard to add new network protocols as well as new cross-layer adaptations.

Our goal is to address these problems by providing a systematic framework that allows the modular implementation of cross-layer adaptations while maintaining to a significant degree the advantages of the layered architecture.

1.1 The Layered Model

The design and implementation of the IP-based Internet is based on the hourglass model, which is an underlying key to the Internet's success. The hourglass model divides the complex tasks of the network into small pieces and defines a set of layers each of which implements some task while leaving other tasks to other layers. Each layer communicates with the other layers through a limited set of interfaces, thus hiding the details of its implementation. The most important interface is

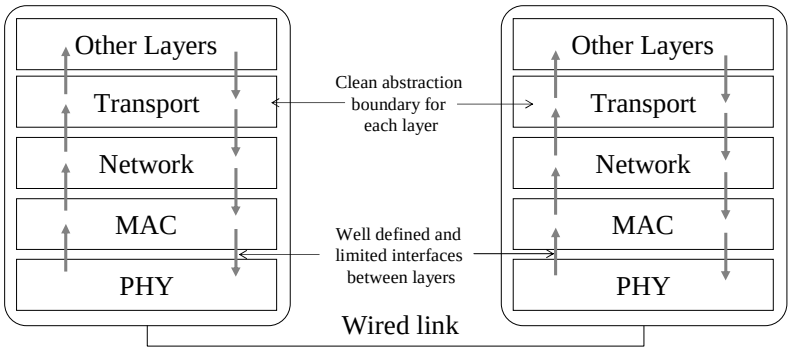


Figure 1.2: The five layers refined from the OSI seven layer model

one that allows a layer to exchange packets with the layers directly above and below it. In addition, limited interfaces are provided to allow a higher layer to read or write some parameters defined in the lower layer, such as the maximum transmission unit (MTU) size defined in the lowest layer. The key point is that the hourglass model defines a clean boundary for each layer and simple interfaces between layers, allowing us to implement each layer without depending on other layers. This modular implementation allows us to change the functionality of a layer without affecting other layers and thus helps us to manage complexity in the design and implementation of the very large distributed hardware and software artifact that comprises the Internet.

Although the Internet is based on the hourglass model, for our purposes it is more useful to consider another layered model of networks, the Open System Interconnection (OSI) seven-layer model [6], which we simplify to five-layers to explain the current implementation of the Internet. Fig 1.2 shows the five layers and how each layer communicates with the other layers.

The lowest layer is the physical (PHY) layer. The PHY is responsible for the actual transmission of raw bits over a communication medium such as a wire or a wireless radio frequency (RF) channel. Thus the PHY mediates between the analog "signals" of the physical world and the digital world. Using a variety of

signal processing technologies, the PHY encodes data to be transmitted into analog signals and decodes the received signals into digital data.

The next layer is the data link/medium access control layer, which is usually referred to as the MAC. The MAC is responsible for reliable communication between two peer nodes that are directly connected to each other. Thus the MAC coordinates the transmission of network nodes that share a communication medium to allow a node to transmit a packet without interference from other nodes. In addition the MAC uses a set of error control techniques such as automatic repeat request (ARQ) which supports reliable transmission by retransmitting a dropped packet.

The next layer is the Network layer, which is responsible for managing the multihop paths between two network nodes by connecting the individual links created by the MAC. To discover and maintain multihop paths through which a source node delivers packets to a destination node, a routing protocol in the Network layer builds a network topology and determines a set of intermediate nodes that forward packets to the destination node.

The next layer is the Transport layer, which is responsible for supporting end-to-end communication along the paths created by the Network layer. One main concern of this layer is to guarantee delivery of packets, and thus many reliable transport protocols like the transmission control protocol (TCP) provide a reliable end-to-end communication channel using the inherently unreliable connections provided by the Network layer.

Finally we simplify all the higher layers that lie on the top of Transport layer to "other layers", since in this dissertation we are mainly concerned with the aforementioned four layers in the OSI seven layer model. For our purpose, the key is that these layers communicate with another network node along the logical end-to-end channels that the Transport layer creates.

As in the hourglass model, the layered architecture provides a clean abstrac-

tion boundary for each layer. As Fig. 1.2 shows, a layer only communicates with the layers above and below it. We refer to the layers that make up the network as the "protocol stack".

1.2 Wireless Networks

For networks made up of reliable and static wired links such as most of the Internet, the layered architecture is remarkably successful, and its clean abstraction boundaries and interfaces work well. However, the introduction of wireless links has revealed that the abstractions are not as cleanly defined as one might hope and that the interfaces between layers are not as simple as one might expect. The classic example is TCP running over wireless links [7, 8]. TCP is one of the main Transport layer protocols that support a reliable connection between two ends by retransmitting a dropped packet [9]. TCP assumes that packet drops are caused by network congestion rather than transmission errors on the link. Thus, after a packet drop, TCP cuts down the sending rate to reduce the load. In fact, if this packet is dropped due to a transmission error of a lossy wireless link, TCP needs to keep retransmitting the packet without reducing the sending rate, to increase the chance of delivery. To solve this problem, TCP needs to communicate with the MAC, which resides several levels down the stack, to find the exact reason for the packet drop. This is just one example where there needs to be enhanced communication across the layers, but which is not supported by the conventional layered architecture.

In general, wireless links differ from wired ones in a number of important ways. The properties of wireless links such as error rate, bandwidth, and latency change dynamically. Further, many properties can be changed by controlling the PHY. Thus wireless links introduce a variety of new interactions across the layers. For example, consider transmission power control, a PHY property. When the power level changes, the set of neighbor nodes with which a node can directly communicate

may change. Thus the MAC might control which neighbor nodes are used for its single hop communication by changing the transmission power. Further, the Network layer might want to change the transmission power to change the multihop path to a destination node. Changing the power controls the neighbor nodes used for single hop communication and thus changes the network topology that is used to discover a multihop route. Power control requires a new way of interacting across the layers. The Network layer needs to communicate with the PHY, which resides several levels down the stack. Even the MAC needs new ways of interaction to communicate with the PHY, which is directly below it, since the conventional layered architecture does not support such enhanced communication even between adjacent layers.

1.3 The Cross-layer Model

The introduction of wireless links has created a need to optimize the performance of the networks by introducing enhanced communication across the layers. This is "cross-layer" protocol design or adaptation (we will refer to it as cross-layer adaptation). In fact, cross-layer adaptation has become a very active research area [2, 3, 4, 5], and it has been shown that this approach can solve a number of problems caused by the introduction of wireless links.

Fig. 1.3 shows many of the possible interactions that can occur when we implement cross-layer adaptations using the conventional layered architecture. The key problem is violating the fundamental structure of layering, i.e., the clean abstraction boundaries and the simple interfaces that allow the modular implementation of each protocol layer. An adaptation process at one layer may need information from any other layer including nonadjacent ones. Further, information may need to be exchanged with layers on other nodes. For example, the Network layer of a source node can require the knowledge of error rates of all the wireless links in a

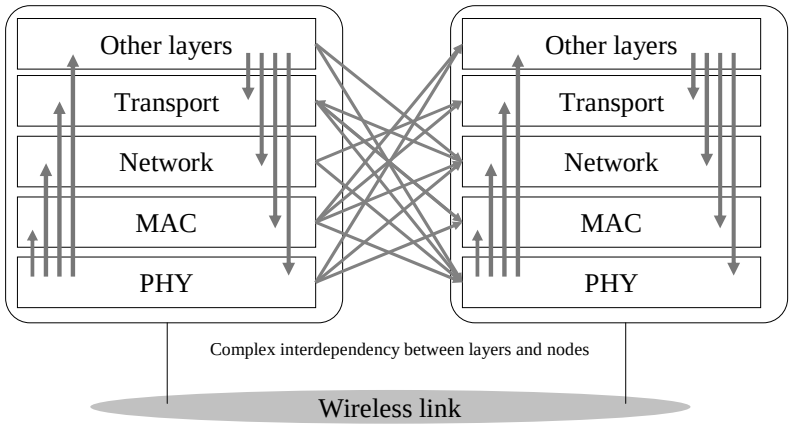


Figure 1.3: Cross-layer communication paths

network to find a robust multihop path to the destination. Thus the implementation of cross-layer adaptations may introduce almost arbitrary interdependency between layers. Further this complex interdependency can turn the layered architecture into a monolithic chunk of spaghetti code [10, 11, 12]. The result compromises the modularity of the layered architecture, which prevents wireless systems from being maintainable.

Unfortunately, most of the work on cross-layer adaptations has proceeded in an undisciplined way and has disregarded the design and implementation advantages of the layered architecture. These ad-hoc cross-layer designs and implementations, which fundamentally tailor the existing network implementations to their own needs, not only compromise the advantages of modularity found in the layered architecture, but also increase the complexity of implementation of the cross-layer adaptations themselves. For example, to implement an adaptation process modifying a protocol layer, we need knowledge of the underlying protocol implementation and introduce significant change to the existing implementation. Further, an implementation that is tailored to one particular system typically may not be used in other systems.

We address the problems that arise when we implement cross-layer adap-

tations in an ad-hoc manner by providing a software architecture that allows the systematic and modular implementation of cross-layer adaptations while maintaining to a significant degree the clean abstractions and interfaces found in the underlying layered architecture. Thus far, several software architectures have been proposed [13, 14, 15, 16, 12, 17] to address the same problems. However, to our best knowledge, there has been essentially no general consideration of how to implement cross-layer adaptations in a systematic way starting from understanding the complex and wide design space of cross-layer adaptations.

1.4 Thesis

Our thesis is:

> A new software architecture to support cross-layer design in wireless networks can provide a useful framework that supports the systematic implementation of a wide variety of cross-layer adaptations. This systematic framework can reduce the complexity of implementation of cross-layer adaptations by maintaining the advantages of modularity found in layered network architectures.

1.5 Goals and Approaches

Our goal is to demonstrate that a new software architecture is able to support the implementation of a wide variety of cross-layer adaptations while maintaining the advantages of modularity found in a layered network architecture. Further, we aim to demonstrate that such a systematic framework can reduce the complexity of implementation of cross-layer adaptations. We divide the work required to achieve the goal into two major steps. In the first step, we are mainly concerned with the development of a new software architecture, and in the second step we are concerned with the evaluation of the architecture. To show our approach in more detail, we

present the detailed steps required to develop the architecture and the key strategy to evaluate the architecture.

1.5.1 Architecture Development

In the first major step demonstrate that a new software architecture can support the implementation of a wide variety of cross-layer adaptations while maintaining the modularity of a layered network architecture. There are three detailed steps for developing such an architecture, and the work of each step is presented as an independent chapter.

The first step is to explore the design space for cross-layer adaptations. We have developed a taxonomy that generalizes and classifies a wide variety of cross-layer adaptations. Our taxonomy allows a precise description of cross-layer adaptations and gives us a common language to describe and analyze possible designs. Moreover, our taxonomy serves as a framework that can be used for creating our software architecture.

The second step is to create a software architecture that supports the implementation of a wide variety of cross-layer adaptations described by our taxonomy. Our strategy for developing such an architecture is actually to create two architectures, a "conceptual" one, followed by a "concrete" one. In this step, we have first created a conceptual architecture. A conceptual architecture is one type of system description method that shows us a set of components and their relationships that are "meaningful" to understand a system at a high-level of abstraction [18]. Our conceptual architecture shows the key mechanisms that are required to support systematic implementation of cross-layer adaptations and thus helps us to understand how we can implement cross-layer adaptations using our framework. Moreover, our conceptual architecture is designed to serve as a generic cross-layer architecture [19]. Thus our architecture describes the key components that are required to support

cross-layer adaptations and allows us to derive a wide set of cross-layer architectures.

The third step is to create a concrete architecture. A concrete architecture shows more detailed properties of individual components and relationships that can occur when we implement a system. We have created such a concrete architecture based on Hydra [20]. Hydra is a flexible wireless network testbed that allows us to experiment with a variety of wireless protocols using a real wireless environment. Thus our concrete architecture shows how we can map our conceptual architecture to Hydra and presents more detailed issues that can arise when we implement our framework on a real wireless system.

A further goal of this step is to show how our conceptual architecture serves as a generic cross-layer architecture, which allows us to derive and describe a wide set of cross-layer architectures. Creating a concrete architecture validates that our conceptual architecture can be used to derive a cross-layer architecture for a real wireless system. Further we have validated our architecture by presenting some existing cross-layer architectures whose goal is also to provide a systematic framework that can be used to implement a wide variety of cross-layer adaptations. We have analyzed the mechanisms provided by those existing architectures using our architecture and thus demonstrated that our architecture can describe a wide set of cross-layer architectures.

1.5.2 Evaluation of Architecture

In the second major step, we evaluate the proposed architecture to show how the architecture allows a systematic and modular implementation of cross-layer adaptations. The key strategy for our evaluation is to perform a set of case studies, each of which implements a cross-layer adaptation within Hydra based on the proposed concrete architecture. As case studies, we implement three cross-layer adaptations, cross-layer rate control, cross-layer contention window control and a link-aware rout-

ing protocol. To show how we use the three cross-layer adaptations to evaluate our architecture, an overview of our evaluation is first presented as an independent chapter. We then present each case study as an individual chapter.

There are three major goals we aim to achieve by the case studies. The first goal is to implement and evaluate the key mechanisms provided by the proposed architecture. We choose three cross-layer adaptations each of which requires a fundamentally different set of mechanisms provided by our architecture. Thus implementing the three adaptation processes allows us to evaluate the key components of our architecture. The second goal is to evaluate how our architecture supports the implementation of variety of cross-layer adaptations while maintaining the modularity of the existing protocol implementations. The final goal is to evaluate the performance of our architecture. To achieve these later two goals, in addition to implementing the adaptations based on our architectural framework, we also implemented the same adaptations in a conventional way in which a cross-layer adaptation directly communicates with other layers. In each case study, we compare both implementation techniques and show the benefits and drawbacks of using our architecture.

1.6 Road Map

The remainder of this proposal is organized as follows. We begin in Chapter 2 with the creation of a taxonomy that describes the design space of possible cross-layer adaptations. In Chapter 3, we develop a conceptual architecture that defines the high-level software structure and its components. In Chapter 4, we refine the conceptual framework into several concrete designs that are to be implemented on working wireless network systems, in particular the Hydra testbed. In Chapter 5 we present an overview of how we evaluate the architecture using the three case studies. Chapter 6 shows the case study which implements and evaluates the architecture

using cross-layer rate control. Chapter 7 presents the case study of cross-layer contention window control. Chapter 8 shows the case study of a link-aware routing protocol. Chapter 9 concludes with contributions of our work and future work.

Chapter 2

Taxonomy

The design space of possible cross-layer adaptations is broad and complex. To define the range of concerns and to control complexity, we have developed a taxonomy that structures this space. Our taxonomy allows a precise description of cross-layer adaptations, and more importantly, gives us a common language to describe and analyze our problem space. Moreover, serving our ultimate goal, it provides a framework for a software architecture that supports the implementation of a wide variety of cross-layer adaptations.

Our goal is to develop a taxonomy that captures a wide variety of adaptations. However, our strategy for its creation is to introduce a few cross-layer adaptation examples and use them to create the hierarchy. Since the motivating examples are simple but cover a broad range of the problem space, this approach significantly reduces the difficulty of developing the complete taxonomy. Based on our examples, we generalize the detailed operations, classify them into types, and then incrementally add the generic types to the taxonomy.

Fig. 2.1 shows the three top level elements of our taxonomy.

- Information: data that drives cross-layer adaptation,

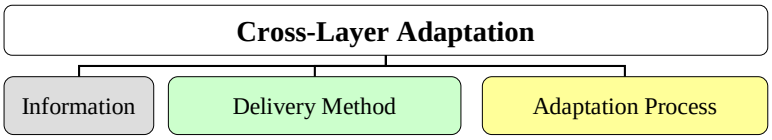


Figure 2.1: Basic elements that comprise a cross-layer adaptation

- Delivery method: mechanisms used to communicate this data,

- Adaptation process: the actual adaptation processes themselves.

Our examples will show how these three elements can be used to describe the key aspects of cross-layer adaptation. Further, examples will be used to build the taxonomy upon this basic structure.

2.1 Motivating Examples

We will use a few examples of cross-layer adaptation to motivate our taxonomy. These motivating examples are general enough to allow us to gain insight about our problem space and to develop a taxonomy that describes a wide variety of cross-layer adaptations. Further, they are simple enough to efficiently deal with the complexity of developing the taxonomy. Our two main examples are:

- Cross-layer rate control: To maximize the wireless channel utilization, the MAC adaptively changes the data rate for transmitting packets based on channel status information from the PHY.

- Cross-layer protocol reconfiguration: To cope with heterogeneity in the wireless communication environment, a manager autonomously reconfigures the protocol stack by using monitoring information reported by the PHY.

A further example is an extension of cross-layer rate control. It is based on cross-layering between different nodes and allows us to complete our example-based taxonomy.

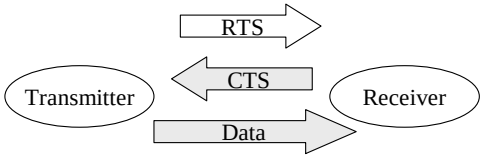


Figure 2.2: Basic operation of IEEE 802.11 DCF system

2.1.1 Cross-Layer Rate Control

Our first example is cross-layer rate control, which utilizes cross-layer interactions between the MAC and PHY to maximize utilization of the wireless channel [21, 22, 23, 24, 25, 26, 27, 28, 29, 30]. The role of our first example is two fold. It introduces a base structure for our taxonomy that can be easily augmented by the motivating examples. Then, it shows how to classify the detailed operations in the example and how to add the new classifications into this base structure.

The opportunity for rate control is that variations in the wireless channel imply variation in the maximum data rate for transmission. When the channel status is good, a higher data rate increases the channel utilization by speeding up transmission. However, as the channel status gets worse, a high data rate can decrease utilization, since errors become more common. A solution is adaptive rate control, which controls the rate after observing the wireless channel. Such an adaptation requires cross-layering between the MAC and the PHY due to a deficit of information in each layer. The MAC is aware of the properties of each packet such as its destination and length that impact proper data rate selection, but it does not have the channel information. In contrast, the PHY is aware of the variations in the wireless channel, but it does not have the packet level information of the MAC. Cross-layering enables information exchange between these two layers and thus facilitates achieving rate control.

Background

When the channel changes quickly, it is appropriate to choose the proper data rate for each data transmission by using up-to-date channel status information. The goal of our cross-layer rate control is such a packet-by-packet rate control that opportunistically utilizes the wireless channel to the greatest extent possible. The first cross-layering required is the movement of channel status information from the PHY to the MAC, which enables rate control to be performed in the MAC. The second cross-layering happens when the MAC informs the PHY of the selected data rate, which enables proper signal processing and transmission in the PHY.

To be concrete, Fig. 2.2 shows the IEEE 802.11 distributed coordination function (DCF) MAC [31]. It is based on carrier sense multiple access with collision avoidance (CSMA/CA) in which a node clears the area of possible interferers before a data transmission. To achieve this, the transmitter MAC sends a request-to-send (RTS) message just before a data transmission. Then, the receiver responds with a clear to send (CTS) message, which triggers the data transmission.

For our rate control protocol, the CTS message also serves as a probe to measure the channel status. Under the assumption that the channel status from a transmitter to a receiver is the same as that from the receiver to the transmitter, the channel status measured at the PHY when it receives the CTS has a strong time correlation with the actual channel that the data will experience. In such a symmetric channel, when the MAC receives a CTS, it can adjust the data rate for each data packet by obtaining up-to-date channel information from the PHY.

After calculating the data rate, the MAC informs the PHY of the current data rate. Since the rate information must be available at the PHY at the time of data transmission, the MAC delivers both the information and the packet itself all at once. In more detail, the MAC concatenates the rate information onto the data packet and sends the concatenated packet to the PHY. Then, the PHY separates

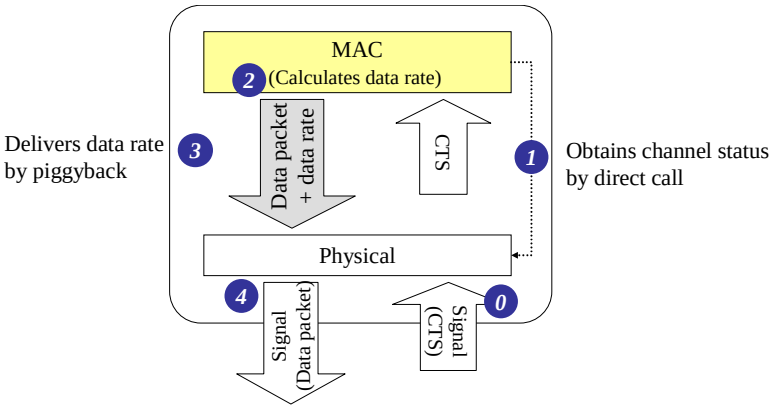


Figure 2.3: Operation of the cross-layer rate control

the rate information from the packet and uses it to properly encode the remaining data packet into the signal. Such 'piggybacking' is one typical way of synchronizing any additional information with its corresponding packet.

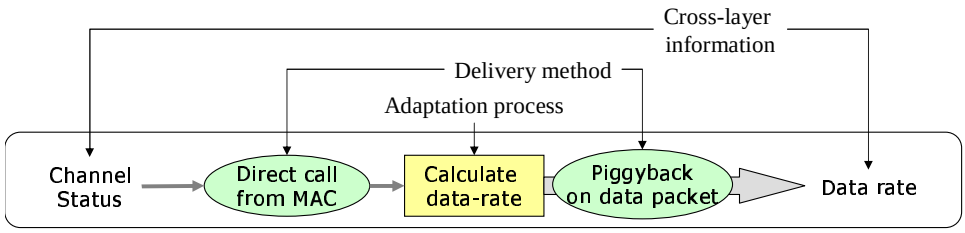
Operation

Fig. 2.3 shows the detailed operations that realize cross-layer rate control. The steps are:

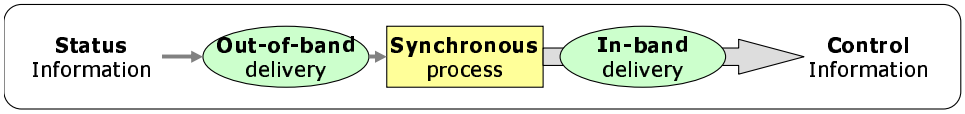
Step 0: When the PHY detects a signal, it estimates the channel to decode the signal. After decoding, it saves the channel status information and sends the packet to the MAC.

Step 1: When the packet that the MAC receives is a CTS message (which implies the MAC already has a data packet ready to transmit), the MAC makes a direct call to pull the channel status information from the PHY.

Step 2: The MAC selects a data rate for the data packet. The details depend on the exact control algorithm used. Our only concern here is that the MAC somehow maps the estimated channel status into the data rate for transmitting the packet.



(a) The basic elements at each operation



(b) The generic types at each operation

Figure 2.4: Sequential flow of the cross-layer rate control

Step 3: The MAC layer informs the PHY layer of the proper rate by piggybacking the rate information on the actual data packet.

Step 4: The PHY separates the piggybacked rate information from the data packet, encodes the data packet into the signal using the rate information and then transmits the signal.

Elements

Building a taxonomy based on examples requires classifying the implementation-specific operations. Fig. 2.4(a) serializes the operations of our rate control protocol to facilitate classification and shows how each operation is mapped into our basic high-level elements. The channel status and data rate are pieces of cross-layer *information* that are exchanged between the MAC and PHY. The direct call from the MAC and piggybacking of information on the data packet are *delivery methods* that enable information exchange. Finally, the calculation of the data rate in the

MAC is the cross-layer *adaptation process* itself.

As the next level of classification, Fig. 2.4(b) generalizes the operations into generic types based on the detailed characteristics of each operation. Such types are important because they show us the significant characteristics of various operations that are used when we realize cross-layer adaptation and allow us to choose the proper operations when we design a new cross-layer adaptation.

Cross-layer information is classified into two generic subtypes, *status* and *control* information. Specifically, the channel status is *status* information whose main role is to monitor the condition of the channel. In contrast, the selected data rate is *control* information that the MAC uses to control the PHY.

Delivery methods also introduce two subtypes, which are *out-of-band* and *in-band* delivery. The direct call from the MAC is an *out-of-band* delivery, since it creates a path that is dedicated to the channel status information and separated from packet delivery. Piggybacking the data rate on the data packet is *in-band* delivery, since it directly utilizes the packet and its delivery path as the method of the data rate delivery.

Finally, the adaptation process introduces one subtype, *synchronous* adaptation. The calculation of the data rate is a *synchronous* adaptation, since it begins after the MAC receives the CTS and ends before the MAC sends the data packet. To achieve opportunistic control by using up-to-date status information, the time of such synchronous adaptation should be synchronized to that of packet processing.

Adding to the Taxonomy

Fig. 2.5 shows our taxonomy and how the generic types found in the example fit into that structure. The hierarchy starts with our three basic elements. Each basic element is augmented by new categories, each of which has its own classification criterion and associated generic types.

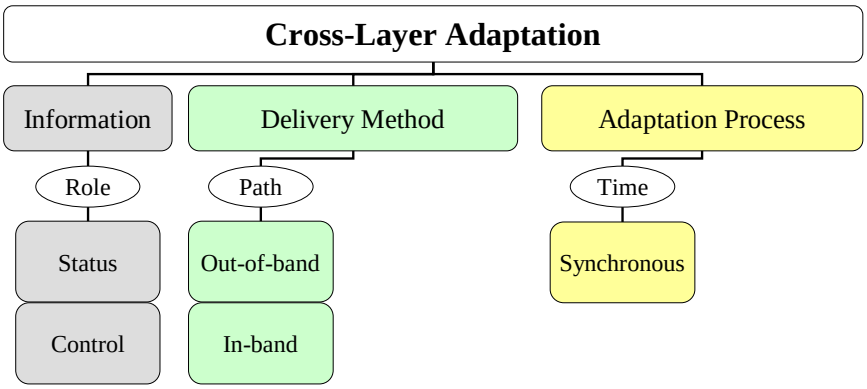


Figure 2.5: Taxonomy built by the cross-layer rate control

The rate control example introduces three new categories for each of the three basic elements. For cross-layer information, the distinction we introduce is the *role* of the data. It concerns how the data is used by the adaptation process and the resulting two new subtypes are *status* and *control*. In delivery method, the distinction we introduce is the *path* of the information delivery. Its concern is the path that the information uses and its relation with that of packet delivery, and thus two subtypes are *in-band* and *out-of-band*. Finally, the adaptation process also introduces one distinction, the *time* of adaptation. Its concern is the time synchronicity of the adaptation process with that of packet processing and its first subtype is *synchronous* adaptation. The complementary subtype will be discussed in a later example, cross-layer protocol reconfiguration.

2.1.2 An Extension of Rate Control

Our next example extends our rate control protocol across both the transmitter and receiver. Doing so solves a potential problem with our existing protocol and allows us to extend our delivery mechanisms to the internode case. The potential problem is that the channel status from a transmitter to a receiver may be different than that from the receiver to the transmitter [32]. In such an asymmetric channel, it is

inappropriate to control the data rate of the data packet by using the channel status observed at the transmitter. Our solution is to use cross-layering involving both the transmitter and receiver. When the receiver receives a RTS from the transmitter, it measures the channel status and informs the transmitter. This extension requires a cross-node information exchange in addition to the exchanges needed previously.

Background

The goal of this extension is to obtain more accurate channel status information when there is an asymmetric channel. The first change required is to measure the channel status at the receiver instead of at the transmitter. In the DCF system (Fig. 2.2), the RTS message now serves as a probe for the channel measurement as the CTS did in the previous example. Since the RTS is sent shortly before the data, it experiences a channel that is well correlated with the one that the data will experience.

The core of the extension is how to deliver this status information back to the transmitter, so it can actually perform the adaptation. The CTS message is used to transport this information. The receiver piggybacks the channel status information on the CTS, and the information reaches the MAC in the transmitter just before the data transmission. The rest of the operations are exactly the same as in the previous example.

Operations

Fig. 2.6 depicts the operation of this extended version of rate control. Since measuring the channel status at the receiver is the same as previously except for the location of the operation, the only additional operation is:

> Step 2: The MAC in the receiver piggybacks the channel status information on the CTS message. Such piggybacking actually places the information inside of

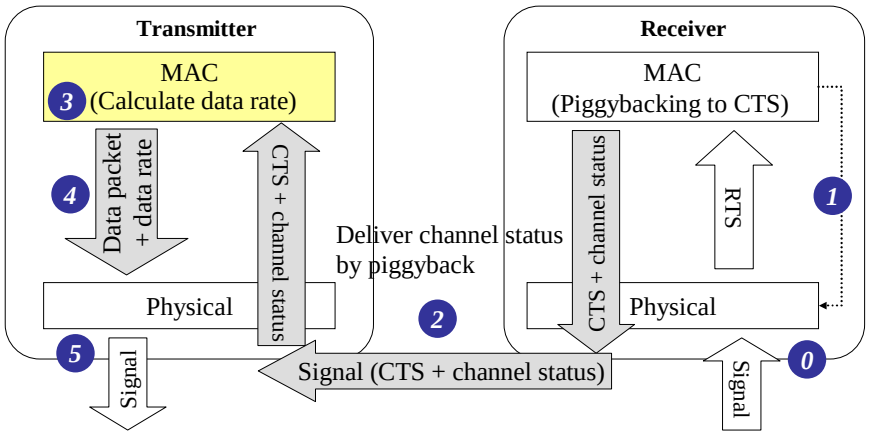


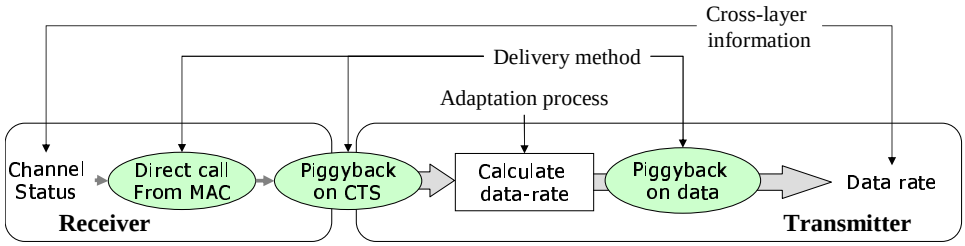
Figure 2.6: Operation of the extended cross-layer rate control

the CTS and makes the information a part of the CTS to allow the cross-node delivery. Then, the PHY in the receiver encodes the CTS into the signal and transmits it to the transmitter. After the PHY in the transmitter receives the signal, it decodes and sends it to the MAC. Among the sub-steps for the cross-node delivery, only the transport of the information from the receiver to the transmitter is a new operation added for this extension.

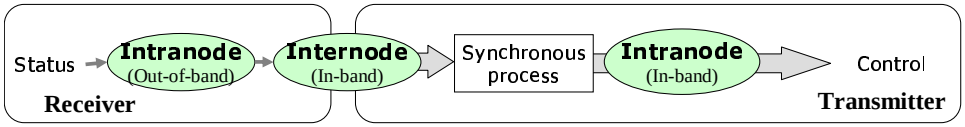
Elements

Fig. 2.7(a) serializes the operations of this example and shows the mapping of high level characteristics into our three basic elements. The piggybacking of the channel status on the CTS, the additional operation for this extension, is classified as a *delivery method*. All other operations are as previously and thus also fall into one of three basic elements.

Fig. 2.7(b) shows the generalization of the operations into the detailed subtypes. Since no change has been made for the *information* and *adaptation process*, no new types are introduced for those elements. However, the piggybacking of the channel status on the CTS causes the introduction of new subtypes for the *deliv-*



(a) The basic elements at each operation



(b) The generic types at each operation

Figure 2.7: Sequential flow of the extended cross-layer rate control

ery method. Since this mechanism allows the channel status information to move between nodes, it is different from the other piggybacking that informs the PHY in the same node and the direct call from the MAC that obtains the information from the PHY in the same node. The main difference here is the *range* of the information delivery with respect to the node as a boundary. Crossing the node boundary requires the use of the wireless channel and thus introduces significantly different issues such as lower bandwidth, frequent packet drops, etc. As a result, the delivery methods are classified into *internode* and *intranode* types.

Adding to the Taxonomy

Fig. 2.8 presents all the generic types used in this example. A new category augments the delivery method and its distinction is the *range* of the delivery. It concerns where the information reaches viewing the node as a boundary and introduces the two new types, *intranode* and *internode* deliveries. These new categories and new types

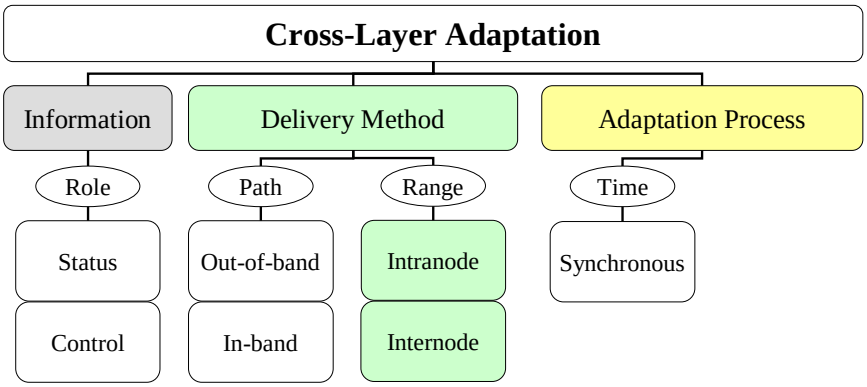


Figure 2.8: Taxonomy expanded by the extended cross-layer rate control

are orthogonal to the preexisting category and its *out-of-band* and *in-band* types resulting in four kinds of delivery methods. Such combination allows us to capture both the range and path of a delivery method. For instance, the piggybacking of the channel information on the CTS is the internode delivery using the in-band connection.

2.1.3 Cross-Layer Protocol Reconfiguration

Our final example, cross-layer protocol reconfiguration, has a different processing style from our previous examples and thus allows us to extend our taxonomy to new types of adaptation. The main difference is the timing of the adaptation process, which is asynchronous to packet processing. For example, suppose that we want to allow a mobile device to move from an IEEE 802.11 wireless network to a Bluetooth network. Such a device needs to be able to switch protocols at run-time. One approach is autonomous reconfiguration of the protocol layers by active monitoring of the wireless communication environment. Essentially, the node monitors the communication environment and switches the protocols when it detects a different protocol packet in use. Because the adaptation is not coordinated with the arrival or departure of a packet, we view it as an asynchronous process.

Background

The goal of cross-layer protocol reconfiguration is autonomous reconfiguration of the protocol layers to cope with heterogeneity in the wireless communication environment. Assuming the protocol modules are flexible enough to support run-time reconfiguration of the layers [33, 34, 35], two additional operations are required to meet this goal. One is monitoring the communication environment, and the second is managing the actual reconfigurations. Thus, cross-layering serves as the glue for the overall adaptation by allowing the information exchange between the various operational entities.

In more detail, the PHY hears all the signals in the wireless environment and detects the changes in the communication standard. In the multi-standard environment, such monitoring is the base requirement for the PHY itself to find the proper signal processing methodologies and to decode the various waveforms defined by each standard. In this example, this detection also serves as a trigger for the protocol reconfiguration. When the PHY detects a change, it informs the reconfiguration process about the new standard.

After notification, the actual reconfiguration is coordinated outside of the protocol processing modules. Since the adaptation process requires global information about the protocol layers, using a global manager makes it easy to manage all the configuration parameters for all layers. Further, since the manager is not part of the protocol processing modules, it can change any layers and is not affected by such changes.

This adaptation process introduces the two cross-layer interactions. First, the PHY informs the global manager that a new standard was detected. Second, after finding the proper configuration for the new communication environment, the manager sets the configuration parameters for all the selected layers. If we assume that, for simplicity of explanation, the new environment requires reconfiguration

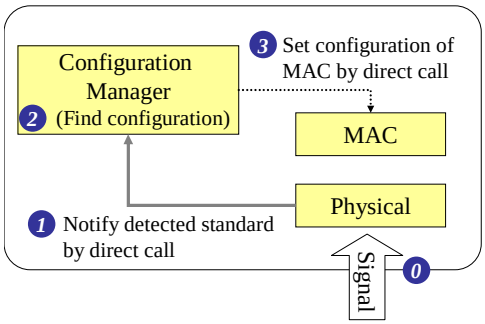


Figure 2.9: Simple flow diagram for cross-layer protocol reconfiguration

of only the MAC, this example becomes a cross-layer adaptation that changes the behavior of the MAC by using the information from the PHY.

Operation

Fig. 2.9 shows each step of the cross-layer protocol adaptation process:

Step 0: Whenever the PHY detects a signal in the wireless medium, it finds the proper signal processing methodology to decode the signal.

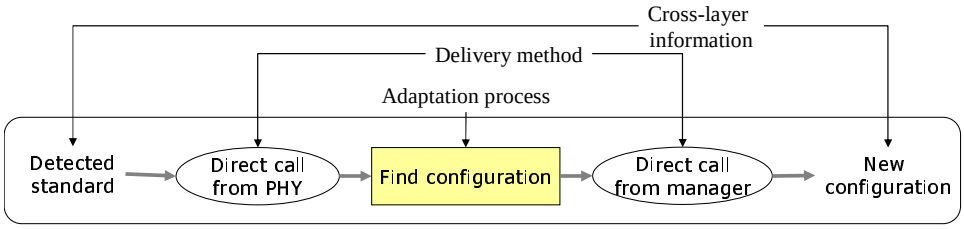
Step 1: If the PHY detects a change of the standard, it sends the detected standard information to the configuration manager by direct call.

Step 2: The configuration manager finds the proper configurations of protocol layers that meet the new standard. In this specific scenario, the result requires only the reconfiguration of the MAC.

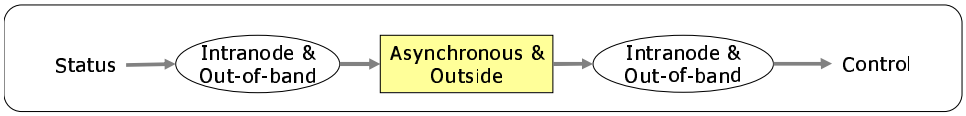
Step 3: The configuration manager changes the configuration of the MAC layer by direct call.

Elements

Fig. 2.10(a) shows how the basic operations are mapped into our high level categories and thus shows that the example also conforms to the proposed three basic elements.



(a) The basic elements at each operation



(b) The generic types at each operation

Figure 2.10: Sequential flow of the cross-layer protocol reconfiguration

The detected standard and new configuration are the *information* and the direct calls are the *delivery methods*. Finally, finding the configuration is an *adaptation process*.

Fig. 2.10(b) shows the mapping of the operations into the generic types. The detailed operations for the *information* and *delivery methods* are mapped into the preexisting subtypes. The newly detected standard is *status* information and the new configuration is *control* information. The direct calls enable *intranode* delivery of data between the manager and the PHY and also between the PHY and the manager by using *out-of-band* connections.

However, the asynchronous aspect of the adaptation process is not captured by any preexisting subtypes. Here, the manager starts the reconfiguration process triggered by a notification from the PHY, in contrast to the rate control scenarios in which the MAC initiates the data rate selection at the time of packet reception. More importantly, this adaptation can still achieve its goals even if it delays the time of actual reconfiguration, while the rate control needs to finish each adaptation process before each data transmission. As a complementary type to *synchronous*

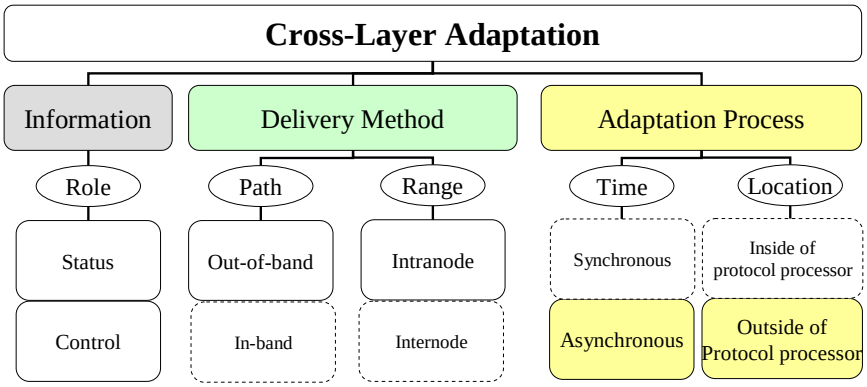


Figure 2.11: Taxonomy expanded by the cross-layer protocol reconfiguration

adaptations, which should be done at the *time* of packet processing, this is an *asynchronous* adaptation, which can be done at the *time* of the certain event that is not tightly related to packet reception.

A significant additional difference is that the adaptation process occurs outside of the protocol processing elements, in contrast to the previous example which executes the adaptation process in the MAC layer. Therefore, the *location* of the adaptation process can be classified into *inside* and *outside* of "protocol processor". The "protocol processor", in our context, represents all the processing entities which are in the middle of the packet delivery path and that handle the protocol packets.

Adding to the Taxonomy

As shown in Fig. 2.11, cross-layer protocol reconfiguration introduces a new distinction whose concern is the *location* of the adaptation process and also shows two new subtypes for the adaptation process. The distinction for the first category was the *time* of the adaptation process and classifies the reconfiguration process as an *asynchronous* adaptation. In addition, the second distinction concerns where the adaptation occurs and generalizes the reconfiguration process as an *outside of the protocol processor* process.

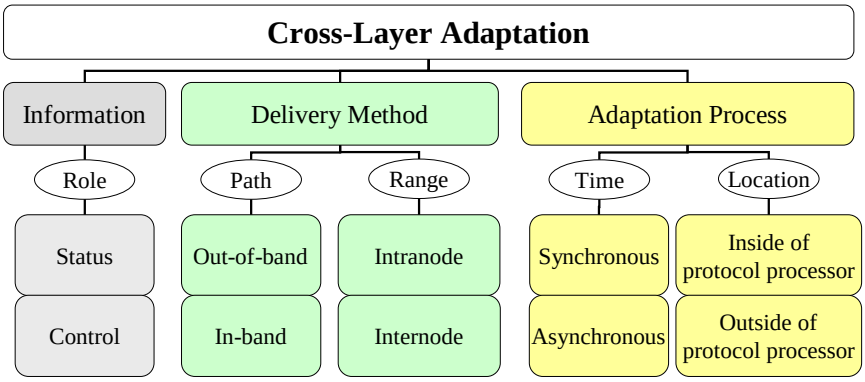


Figure 2.12: A complete taxonomy of cross-layer adaptation

The introduction of these two types requires the two complementary types to complete each category. The aforementioned *synchronous* process type fulfills the classification of the adaptation time, and adaptation *inside of protocol processor* completes the location of adaptation. By combination of all the four kinds of types, we can capture the time and location of the adaptation process. For instance, the rate control example is a synchronous process performed inside of the protocol processor.

2.2 Review and Refinement

Fig. 2.12 shows our complete taxonomy for cross-layer adaptation, after adding all the generic subtypes found in the motivating examples into the proposed framework. To obtain a concrete picture of the taxonomy, we will review each subtype by giving brief descriptions. Further, to allow us to choose the proper mechanisms when we design and realize a new cross-layer adaptation, we will also discuss some of their practical aspects. This detailed discussion also introduces a further refinement of the intranode and internode delivery types.

2.2.1 Cross-Layer Information

The key distinction we introduce for cross-layer information is the *role* of data and two complementary subtypes are *status* and *control*. The role is changed according to the usage of the data during the adaptation process. Therefore, cross-layer information can be *status* information for one adaptation process while it is used as *control* information for some other adaptation.

2.2.2 Cross-Layer Delivery Method

Cross-layer delivery methods are augmented by two independent categories. The first distinction concerns the *path* of the information delivery and the two subtypes are *in-band* and *out-of-band*. The practical characteristic of *in-band* delivery is that it utilizes "existing" protocol packets. In contrast, *out-of-band* delivery creates additional delivery paths dedicated to cross-layer information. This classification is obvious for *intranode* delivery, but also applicable for the *internode* delivery. If *internode* delivery piggybacks cross-layer information on "existing" packets, it is *in-band* delivery. In contrast, if it creates a new packet to deliver information only, it becomes *out-of-band* delivery.

The second distinction is the *range* of information delivery. We view the "node" as a boundary and introduce *intranode* and *internode* delivery subtypes. The reason why the "node" is the important boundary is that information traverses the wireless channel when it is sent to other nodes. *Intranode* delivery does not require the wireless channel and thus provides high bandwidth, high reliability and low latency. In contrast, *internode* delivery requires the wireless channel and thus can suffer from the low bandwidth, frequent packet drops, and high latency.

The *intranode* and *internode* subtypes capture the most significant consideration when we choose the proper delivery methods of cross-layer adaptations. However, it is useful to further refine both the intranode and internode delivery,

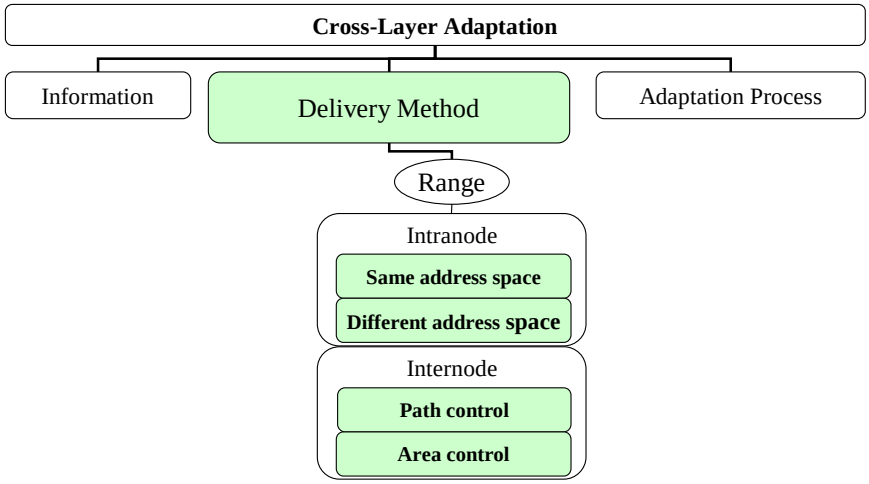


Figure 2.13: Detailed classification of cross-layer delivery methods

since the realizations of both the subtypes introduce additional distinctions that further divide the delivery range.

Refinement of intranode delivery

The realization of *intranode* delivery changes based on the address space in which each protocol layer is running. For example, the wireless MAC protocol is generally running on a network interface card while routing protocols are running on a general purpose processor. To allow the data exchange between those layers, *intranode* delivery now requires interprocess communication mechanisms. Further, when those layers represent data in different ways, it also requires "presentation formatting" which transforms the data into a message format that both layers agree on.

Thus a further distinction for *intranode* delivery is the *address space* of the layers which introduces *same address space* and *different address space* delivery, as shown in Fig. 2.13. The *different address space* delivery needs additional mechanisms than *same address space* delivery. However, it still has the most important characteristics of *intranode* delivery, such as reliability and high bandwidth.

Refinement of internode delivery

Internode delivery is an instance of the *different address space* subtype, since different nodes are inherently different address spaces. However, the more important difference is the use of the wireless channel. The wireless channel makes it more difficult to deliver information because of its dynamic characteristics, but allows information to move to any node in the network by controlling propagation. Fig. 2.13 shows two ways of controlling propagation, *path control* and *area control*.

Path control defines an explicit path to a specific destination node and allows all nodes along a path to generate or consume the information. Such a *path control* enables, for example, a video server to acquire the channel status of all the links to a specific client so that it can adaptively change video resolution. Further, the extended version of the rate control example also utilized path control for its internode delivery. The only two nodes on the path were the transmitter and the receiver, and channel status information was delivered to the transmitter only, not to all nodes that can overhear that information.

In contrast, *area control* only defines the number of hops within which information is propagated and allows all neighboring nodes within that range to obtain the information. Such *area control* enables an *internode* extension of the protocol reconfiguration example so that it cooperates with neighbor nodes. A node can forward the new standard information to its neighbor nodes and thus trigger the reconfiguration process even when they have not yet detected the new standard.

2.2.3 Cross-Layer Adaptation Process

The final element, the cross-layer adaptation process, is augmented by two independent categories. The first distinction concerns the *time* of adaptation and its relation with that of packet transmission or reception. The two subtypes are *synchronous* and *asynchronous* adaptation. In practice, this distinction concerns the

event that triggers the adaptation process. For example, the event that triggered *synchronous* adaptation in the rate control example was the packet reception, and the MAC just used the channel status information at that time. However, if we allow an *asynchronous* extension which re-calculates the data rate only when the PHY reports severe fluctuation of the channel, the triggering event is the channel status itself and is not synchronized to the packet reception or transmission.

The second distinction concerns the *location* of the adaptation process and the two subtypes are *inside* and *outside* of the "protocol processor". Realizing the *inside* adaptation implies that protocol processing deals with both the protocol packet and cross-layer adaptation. Therefore it is useful when the adaptation is tightly coupled to packet processing. In contrast, the *outside* subtype is appropriate when the adaptation requires global information about multiple protocol processors, as the protocol reconfiguration example did. Providing quality of service (QoS) or optimizing energy consumption also involves multiple protocol processors and can select the *outside* subtype for their realization.

Chapter 3

A Software Architecture

The goal for our software architecture is to provide the key mechanisms that are required to implement a wide variety of cross-layer adaptations described by our taxonomy. Our strategy for developing such an architecture is actually to create two architectures, a conceptual one and a concrete one [18]. In general, a conceptual architecture is one type of system description method that shows us a set of components and their relationships that are "meaningful" to understand a system at a high-level of abstraction. Thus in our case, our conceptual architecture shows the key components and their basic operations and presents how we can implement a variety of cross-layer adaptations using our architectural framework. A concrete architecture in general shows more detailed properties of individual components and relationships that can occur when we implement a system. Thus in our case, our concrete architecture shows how we can extend our conceptual architecture to implement our architectural framework on a specific wireless system.

This chapter focuses on the conceptual architecture. Our conceptual architecture [36] introduces a set of key components and their relationships that are required to implement the desired cross-layer adaptations. Thus it helps us to understand how we can implement cross-layer adaptations using our architectural

framework and in practice serves as a reference model from which a variety of concrete architectures can be derived. Further our conceptual architecture shows all the key software components that are required to implement cross-layer adaptations described by our taxonomy. Thus it can describe a wide range of cross-layer architectures (whether they are conceptual or concrete). In that sense, our conceptual architecture is a "generic" architecture [19, 37] from which a wide range of cross-layer architectures can be derived.

We begin by presenting a series of high level goals for our architecture, followed by some key architectural decisions that are motivated by these goals. Then, we use our two cross-layer examples as building blocks to manage the complexity in creating an architecture, as we did for developing our taxonomy. We introduce new software components that are required for implementing each example and flesh out the details of our architecture. Finally, we introduce the components that are not covered by the examples and review the high level properties of individual components and their relationships to manifest the overall architectural design.

3.1 Goals

We present a set of high level goals for our architecture to manifest what we aim to achieve by creating a software architecture. The primary goal of our architecture is to provide a set of mechanisms that can serve to implement the wide variety of cross-layer adaptations described by our taxonomy.

There are a number of secondary goals, which are motivated by a desire to preserve the advantages of the modular layered architecture and further to expand such modularity into the cross-layer adaptations themselves. The first secondary goal is to preserve the modularity of protocol modules. This is key because otherwise we allow arbitrary cross-layering in an ad-hoc manner, thus turning a system into a monolithic chunk, which makes it hard to implement and update individual protocols

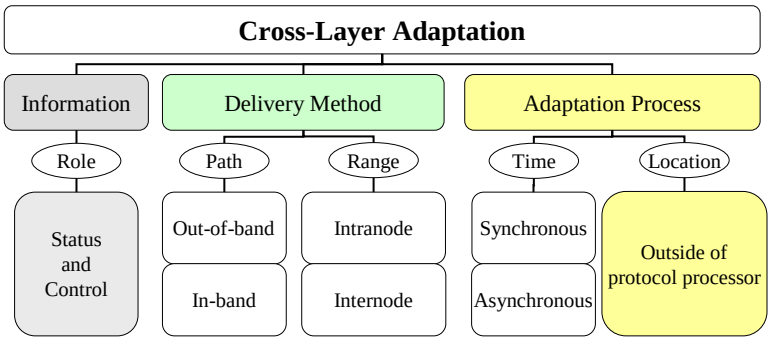


Figure 3.1: A refined taxonomy derived from our architectural decisions

without affecting other protocol implementations. The second goal is to expand this goal for cross-layer adaptation. Introducing a modular implementation to cross-layer adaptation allows us to implement individual adaptation processes as flexibly as possible and further make them portable to a variety of protocol implementations. For example, ideally a new rate control technique can be upgraded based upon the preexisting rate control implementation and can be also moved to another system that has a different protocol implementation.

3.2 Architectural Decisions

The primary goal of our architecture is to provide all the key mechanisms required to implement a wide variety of cross-layer adaptations described by our taxonomy. Some aspects of our taxonomy however cannot be directly applied for developing our architecture and thus we need to refine the taxonomy. Fig 3.1 shows how our taxonomy is refined after we have applied two key architectural decisions. The first decision is simple. Although the classification into status or control information is useful to describe the operations of adaptation, its implementation is the same once the architecture provides a generic representation for cross-layer information. Thus the first refinement is to merge the two subtypes of cross-layer data.

The second and most important decision is the elimination of the *Inside of protocol processor* location of Adaptation process from the cases that our architecture must support. Our secondary goals motivate this decision. If we realize an adaptation as part of a protocol implementation, we require substantial changes to the existing protocol implementation, and also the adaptation is tailored to that implementation. Such changes result in complex interdependency and thus compromise the modularity of not only existing protocols but also the cross-layer adaptation. Thus we decouple the adaptation process from the existing protocol implementation as much as possible and allow the adaptation process to communicate with the protocol layers from outside the protocol module. This minimizes the changes to the protocol implementation and also facilitates relatively independent implementation of adaptations. Thus the key challenge is how our architecture can support the realization of a wide variety of cross-layer adaptations described by our taxonomy, but only from outside the protocol module itself.

3.3 Architecture by Example

To manage complexity in developing an architecture that supports the key mechanisms we need, we use our two previous examples, cross-layer rate control and protocol reconfiguration. We first show the high level architectural changes that are required to implement the adaptation process of each example outside of the protocol module. The rate control example in which the adaptation process was implemented as part of the MAC implementation shows the changes to allow the rate control process to be performed outside of the MAC. Then we introduce the detailed mechanisms required to map the examples to our architecture. Since the examples cover a large part of the design space, they show us most mechanisms required to implement cross-layer adaptations and thus reduce the complexity in developing a complete architecture. For simplicity, we only use the Intranode ver-

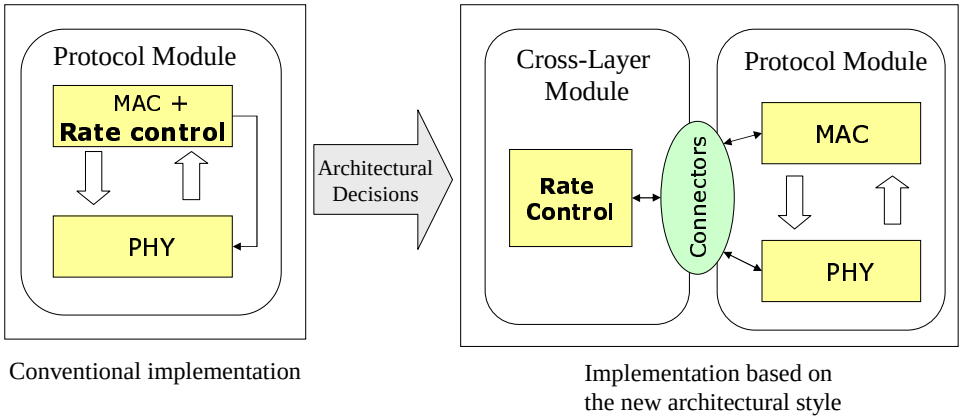


Figure 3.2: Changes of architectural style to map the cross-layer rate control into our loosely coupled architecture

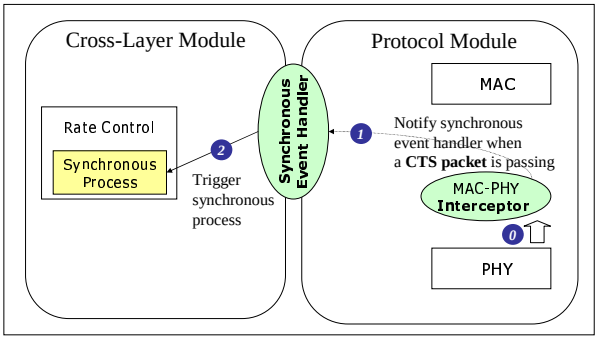
sion of each example, and we will discuss the Internode extension of our architecture in Section 3.4 when we show the complete architecture.

3.3.1 Cross-Layer Rate Control

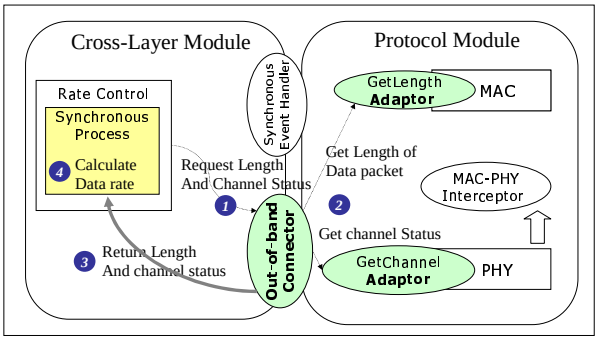
Fig. 3.2 shows the high level architectural style required to implement the rate control process outside of the protocol module. The key change we introduce is decoupling the rate control implementation from the MAC implementation and placing the rate control process in a separate "cross-layer module". Now connectors serve as the bridge between the rate control process and the existing protocol implementations. We introduce a set of detailed mechanisms that are required to allow the rate adaptation to be performed outside of the MAC without substantial change of the existing MAC and PHY implementations.

Architectural Solutions

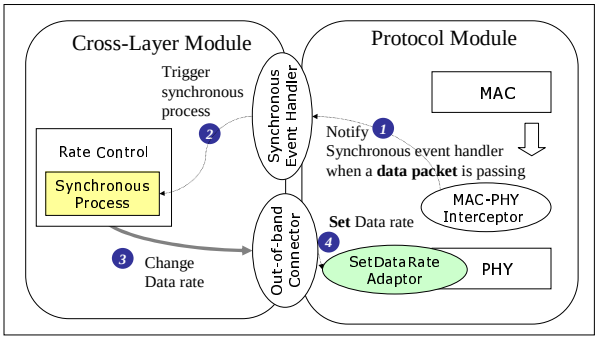
We divide the Intranode version of the rate control process into three high level stages based on the detailed mechanisms required. The key challenge is how each mechanism can allow the rate control process outside of the protocol module to



(a) The first stage that allows the rate control process to be notified of the Synchronous event



(b) The second stage that allows the rate control process to access the out-of-band information



(c) The final stage in which the mechanisms introduced previously are reused

Figure 3.3: The three stages of the rate control process

connect to the protocol module without substantial changes of the existing protocol implementation. Fig. 3.3(a) shows the first stage, in which we introduce the mechanisms that trigger the rate adaptation process outside of the protocol module. The key requirement in this stage is that when a CTS packet moves from the PHY to the MAC, the rate control code must be notified to trigger its synchronous adaptation process. Thus we first introduce an *Interceptor* that mediates the passage of packets between the MAC and PHY. The Interceptor is a shim-layer, which is a layer inserted between two adjacent protocol layers to provide the additional functionality that was not supported by existing protocols. Its implementation conforms to the interfaces of the existing protocol implementations and it can be transparently inserted between two adjacent layers without changing them. Thus the Interceptor does not compromise our modularity goal. In step 1, when the Interceptor has detected a CTS, it notifies the *Synchronous event handler*, which relays the event from the protocol module to the cross-layer module. Then, in step 2, the Event handler notifies the rate control processor, triggering the synchronous adaptation process.

Fig. 3.3(b) shows the second stage, in which we introduce the mechanisms that allow the rate control process outside of the protocol module to obtain information from the protocol modules. In step 1, the rate control process requests the channel status and data length of the packet from the *Out-of-band connector*. The Out-of-band connector allows the cross-layer module to access the information in the protocol module. A difference from the case where the adaptation is part of the MAC is now the rate control process needs information from the MAC as well as the PHY. In step 2, the Out-of-band connector communicates with the GetLength and GetChannel *Adaptors* attached to the MAC and PHY. The Adaptor is a software component that augments the interface of each protocol layer so that the Out-of band connector can obtain the data from the protocol layers. Implementing the Adaptors requires neither detailed knowledge about the underlying protocol imple-

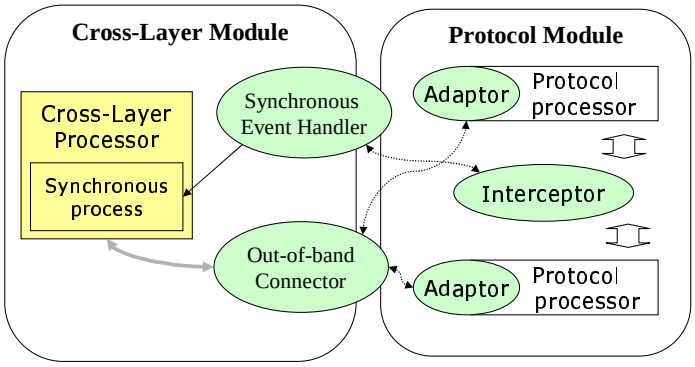


Figure 3.4: Architectural components that allows cross-layer rate control

mentation nor changes to the core functionalities of the existing protocol. Thus the Adaptors simplify (but do not eliminate) changes to the protocol layers. In step 3, the Out-of-band connector returns the information obtained from the Adaptors attached to the MAC and PHY. Then, in step 4, the rate control process calculates the new rate using the information.

Fig. 3.3(c) shows the final stage illustrating how we use the mechanisms we have already introduced to complete the rate control process. In steps 1 and 2, the Interceptor notifies the Synchronous event handler when the MAC sends a data packet, and the Synchronous event handler triggers the synchronous adaptation process again. In steps 3 and 4, the rate control process sets the rates in the PHY using the Out-of band connector and SetDataRate Adaptor attached to the PHY. The only addition is the SetDataRate Adaptor, which allows the Out-of band connector to inform the PHY of the rates.

Adding to the Architecture

Fig. 3.4 shows the architectural components we introduced to implement the rate control example. A cross-layer adaptation is implemented as a *cross-layer processor* in the cross-layer module. Then the *Synchronous event handler* and *Out-of-band*

connector connect the cross-layer processor to the existing protocol module. The Synchronous event handler triggers the synchronous adaptation process of the cross-layer processor when it is notified of a packet passing event, while the Out-of-band connector allows the cross-layer processor to exchange the data with the protocol processors.

The key problem was to allow such connectors to access the protocol module while simplifying changes to the existing protocol implementations. The two components we have introduced to address this problem are the *Interceptor* and the *Adaptor*. Without changes to the existing protocol implementations, Interceptors notify the Synchronous event handler of a packet passing event. Further, since the Interceptor can intercept all the packets that are exchanged between any two adjacent layers, it can access cross-layer information that is piggybacked on a packet and moving along the path of packet delivery. With simple changes to the existing protocol implementations such as augmenting interfaces, Adaptors allow the Out-of-band connector to access cross-layer information that is stored in the protocol processors.

3.3.2 Cross-Layer Protocol Reconfiguration

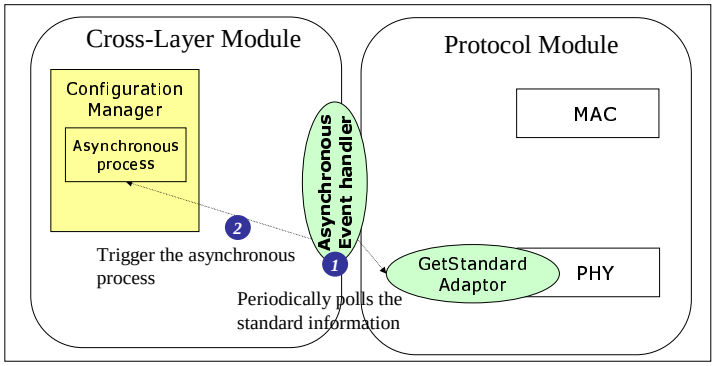
Our next example is cross-layer protocol reconfiguration in which a configuration manager performed the asynchronous adaptation process that reconfigured the MAC protocol when the PHY detects a new communication standard. The configuration manager already performed the adaptation outside of the protocol module, thus there is no change in the basic architecture style. Thus the key concern here is to show how our architecture supports the asynchronous adaptation process outside the protocol module.

Architectural Solutions

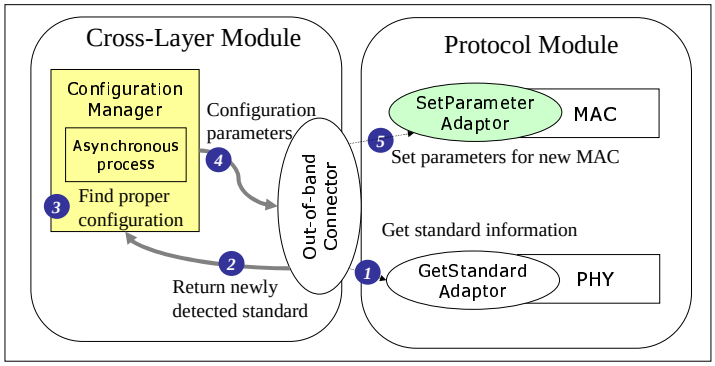
We divide the protocol reconfiguration process into two high level stages based on the mechanisms required. In the first stage, we introduce the mechanisms that trigger the asynchronous adaptation process outside of the protocol module. In the "conventional" way of implementation discussed in Section 2.1.3, when the PHY detects a new communication standard used in the wireless environment, it actively notifies the configuration manager and triggers the asynchronous adaptation process that reconfigures the MAC protocol. This active notification from a protocol layer can be one possible solution that triggers the Asynchronous adaptation process outside of the protocol module. The problem within this mechanism, however, is that the PHY implementation needs to be changed to be aware of the configuration manager, thus making the PHY implementation dependent on the implementation of the configuration manager.

Fig. 3.5(a) shows the mechanisms we introduce to address this problem. In step 1, the *Asynchronous event handler* periodically polls the standards information stored in the PHY to check for a change, instead of the PHY actively notifying the Asynchronous event handler. The polling mechanism only introduces a new Get-Standard Adaptor attached to the PHY to allow the Asynchronous event handler to access the standards information in the PHY and allows the PHY implementation to be independent of the Asynchronous event handler. Although detecting the communication standard used in the wireless environment requires substantial changes to the PHY implementation, the mechanism maintains the modular implementation of the PHY to a significant degree. In step 2, when the Asynchronous event handler finds a change of standard, it triggers the configuration manager.

Fig. 3.5(b) shows the second stage in which the configuration manager executes its adaptation process by using the components we have introduced previously. In steps 1 and 2, the configuration manager requests newly detected standards in-



(a) The first stage that allows the reconfiguration process to be notified of the Asynchronous event



(b) The second stage in which the mechanisms introduced previously are reused

Figure 3.5: The two stages of the reconfiguration process

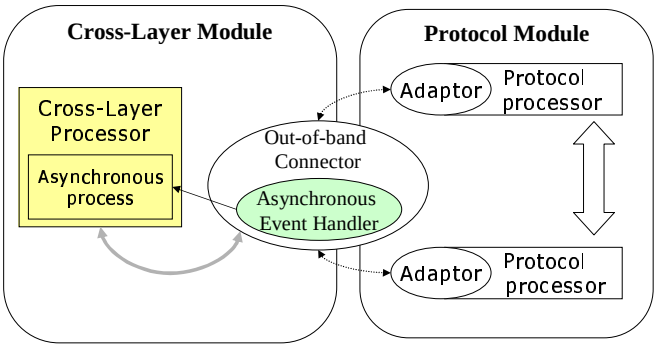


Figure 3.6: Components required for cross-layer protocol reconfiguration

formation through the Out-of band connector and the GetStandard Adaptor. After finding a configuration for the new standard, in steps 4 and 5, the configuration manager configures the MAC by communicating with the SetParameter Adaptor attached to the MAC.

Adding to the Architecture

Fig 3.6 shows the mechanisms used to support cross-layer protocol reconfiguration. The only new component is the Asynchronous event handler. It actively polls the information in the protocol processors by communicating with the Adaptors and triggers the asynchronous adaptation process when it detects a change. Thus the Asynchronous event handler maintains the modularity of the protocol implementation by allowing the protocol processor to be independent of the cross-layer module. In contrast to the synchronous event handler, such a periodic polling mechanism may cause some bounded delay before detecting the changes. However, the asynchronous adaptation process can tolerate such latency according to its definition in our taxonomy.

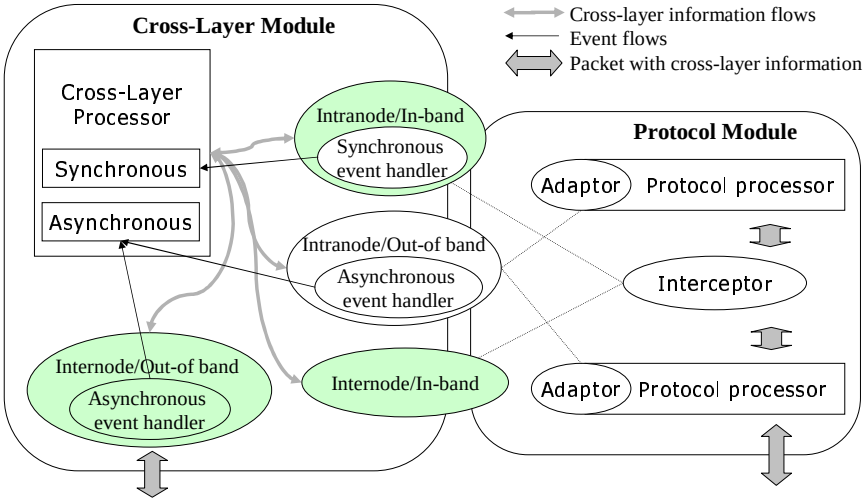


Figure 3.7: A complete conceptual architecture for cross-layer adaptations in wireless networks

3.4 A Complete Conceptual Architecture

Fig. 3.7 shows all the details of our conceptual architecture including the components that were not covered by the examples. To manifest the overall architectural design, we will review the high level *properties* of all the components of our architecture and their *relationships*. We explain each component by grouping them into *data*, *connecting* and *processing* components, which are the three essential components that compose a software architecture [38].

3.4.1 Data Component

The first architectural component is the data component, which implements cross-layer information. Although the data component is not discussed in our examples, one thing we notice from the examples is that the implementations of *status* and *control* information are not different. However, the structure of the cross-layer information required by each adaptation process can be quite different. For example, an adaptation process may require a routing table that is composed of a set of de-

tailed information about the wireless links. Thus the key requirement for the data component is to provide a flexible and extensible data representation that can cover a variety of cross-layer information.

3.4.2 Connecting Components

The second architectural components are the connecting components, which connect the cross-layer adaptation process outside of the protocol module to the existing protocol implementations as well as to the other nodes. All the connecting components in our architecture are typical software connectors [39]. They allow a cross-layer processor to exchange cross-layer data with protocol modules as well as other nodes and trigger the synchronous and asynchronous adaptation processes by delivering the events generated from the protocol module and other nodes.

We first present the new connectors that were not introduced by the examples and flesh out all the details of the connectors. Then we review the high level properties of the connectors by grouping them into *data connectors* which provide the cross-layer processor with the data delivery mechanisms and *event handlers* which trigger the synchronous and asynchronous processes of the cross-layer processor.

New Connecting Components

The new connectors that were not covered by our examples are the Intranode version of the In-band connector and a set of Internode connectors. The first new connector is the Intranode version of the In-band connector, which allows the adaptation process outside of the protocol module to access cross-layer information piggybacked on the protocol packet. The In-band connector is similar to the Synchronous event handler in that it also communicates with Interceptors to access the protocol packets. When an adaptation process requests information piggybacked on a packet, the In-band connector communicates with an Interceptor and accesses information that

moves with the packet from one layer to another. Thus the In-band connector pushes or pulls data in the protocol module's internal data structure that is used to hold an actual protocol packet and additional information corresponding to that packet.

The second new connector is the Internode version of the In-band connector, which allows the adaptation process to deliver cross-layer information to other nodes by using existing protocol packets. As in the Intranode version of the In-band connector, the Internode version of the In-band connector communicates with the Interceptors and accesses data piggybacked on a packet when the packet passes through an Interceptor. The Internode version of the In-band connector however concerns information that is delivered to another node. Thus it places the information inside of an actual packet and makes the information a part of the packet, instead in the data structure that the protocol module uses internally.

The last new connectors that were not covered by our examples are the Internode version of the Out-of-band connector and the Asynchronous event handler. The Out-of-band connector allows the adaptation process to exchange cross-layer information with another node by using additional delivery paths dedicated to the information. Thus when an adaptation process requests the Out-of-band connector to send cross-layer information to another node, the Out-of-band connector creates a new packet that contains only the cross-layer information and sends that independently. Then when an Out-of-band connector in another node receives the cross-layer information, it notifies its Asynchronous event handler. Then, as in the Intranode case, the Asynchronous event handler triggers an asynchronous adaptation process to allow the adaptation process to obtain the information from the Out-of-band connector.

Table 3.1: Basic properties of the four kinds of data connectors

	Intranode		Internode	
	In-band	Out-of-band	In-band	Out-of-band
Location of Information	Packet's internal structure	Protocol layers	Protocol packet itself	Other nodes
Communicating entity	Interceptors	Adaptors	Interceptors	Its counterpart in other nodes

Data Connectors

We have shown all the details of the connecting components provided by our conceptual architecture. Table 3.1 summarizes the basic properties of the *data connectors* of which the main concern is to allow the cross-layer processor to exchange cross-layer data with the existing protocol module and another node. Intranode connectors are used inside a single node to integrate existing protocol modules with our architecture. The In-band connector accesses data in the packet's internal structure when the packet passes through an Interceptor, while the Out-of-band connector communicates with Adaptors to access the data in protocol modules. The Interceptor and the Adaptor are also kinds of Intranode connectors that allow such Intranode connectors to access the data in protocol modules without substantial changes to the existing protocol implementations which are tailored to a particular adaptation.

Internode connectors deliver cross-layer information from one node to another, and thus packets serve as a container for the cross-layer information. The In-band connector piggybacks the data on the existing protocol packet and thus accesses the data when the packet passes through an Interceptor, as the Intranode version does. In contrast, the Out-of-band connector places the data in its own separate packet and sends it independently. Thus it exchanges data with counterparts on other nodes through a separate communication path.

Table 3.2: Basic properties of the four kinds of event handlers

	Intranode		Internode	
	Synchronous	Asynchronous	Synchronous	Asynchronous
Event context	Packet passage	Change of information within protocol module (Periodic polling)	Same with Intranode case	Reception of information from other nodes
Communicating entity	Interceptors	Adaptors	Same with Intranode case	Its counterpart in other nodes

Event Handlers

In our architecture, the synchronous and asynchronous nature of adaptation processes are fundamentally captured by the corresponding event handlers. As shown in Table 3.2, the Synchronous event handler, which covers both the Intranode and Internode cases, triggers synchronous adaptation processes driven by a packet passing event through the Interceptors. In contrast, the Intranode version of the Asynchronous event handler triggers the asynchronous adaptation process when there is a change of information within a protocol module. As a special case, it can periodically poll for information to maintain the modularity of existing protocol implementations. Finally the Internode version of the Asynchronous event handler triggers asynchronous adaptation process when it receives information from other nodes.

3.4.3 Processing Component

We implement the cross-layer adaptations based upon the connectors of our architecture, which provide the key mechanisms that are required to implement cross-layer adaptations while hiding their details from the cross-layer processor. Thus we can implement a variety of cross-layer adaptations as flexibly as possible by using our

architectural infrastructure.

The cross-layer processor is the base unit for the modular implementation of cross-layer adaptations. In our example, a rate control processor implements the synchronous adaptation processes that are required to perform the rate adaptation, and a configuration manager implements the asynchronous adaptation processes required to perform the protocol reconfiguration. In practice however a cross-layer adaptation can be composed of a more complex combination of the processes including both synchronous and asynchronous processes. Further a set of cross-layer adaptations can be run on a single node. Our event handlers decompose such complex combinations of adaptation processes into a series of individual event-driven processes and thus can easily coordinate various combinations of cross-layer adaptation processes in a single system.

Chapter 4

A Concrete Architecture

Thus far our architecture is a "conceptual" architecture, that is to say one that describes at a high-level of abstraction the key mechanisms that are required to support a wide variety of cross-layer adaptations and thus which helps us to understand how we can implement a variety of cross-layer adaptations using our architectural framework. Here, we create a concrete architecture by extending the conceptual architecture and showing the detailed issues that arise in implementing our architectural framework on a real wireless system. We create our concrete architecture based on our wireless network testbed, Hydra [20]. Hydra is a flexible platform that allows us to experiment with a variety of wireless network protocols using a real wireless environment and node implementations. Thus our concrete architecture shows how we can extend our conceptual architecture to allow us to implement a wide variety of cross-layer adaptations within Hydra and describes in more detail issues that can arise when we implement our framework on a real system.

This chapter also presents two cross-layer architectures as related work, among those proposed in the literature [14, 13, 16, 40, 41, 42, 43, 44, 45, 15]. We analyze the mechanisms provided by the existing cross-layer architectures by using our conceptual architecture. Thus, in addition to creating a concrete architecture

by extending our conceptual architecture, describing the existing cross-layer architectures using our conceptual architecture allows us to validate how our architecture serves as a "generic" one that can describe a wide set of cross-layer architectures.

We begin by presenting the details of the Hydra system, followed by capturing the key aspects of Hydra as a target wireless system for our concrete architecture. Then we show how our concrete architecture extends our conceptual architecture to allow us to implement a variety of cross-layer adaptations and consider detailed issues that can occur when we implement the concrete architecture. Finally, we discuss two existing cross-layer architectures and analyze them using our architecture.

4.1 Hydra

We develop our concrete architecture based upon Hydra [20]. Hydra is a multihop wireless network testbed, which is designed to experiment with a variety of novel wireless network protocols in a real wireless environment as opposed to just using simulation. Thus Hydra is composed of flexible hardware and software that allow us to rapidly implement new ideas in wireless network protocols, as well as in the PHY layer. We illustrate the details of the Hydra system and then capture the key aspects of Hydra as a target wireless system for our concrete architecture.

4.1.1 System Configuration

Fig. 4.1 shows the system configuration of a single Hydra node including its flexible hardware and software. Each node is composed of a host system that implements all the software protocols and a set of universal software radio peripheral (USRP) boards [46] each of which implements a radio frequency (RF) front end. The host system is connected to the USRP boards over a universal serial bus (USB) 2.0 connection that allows high-speed serial communications between the host system and the USRP boards.

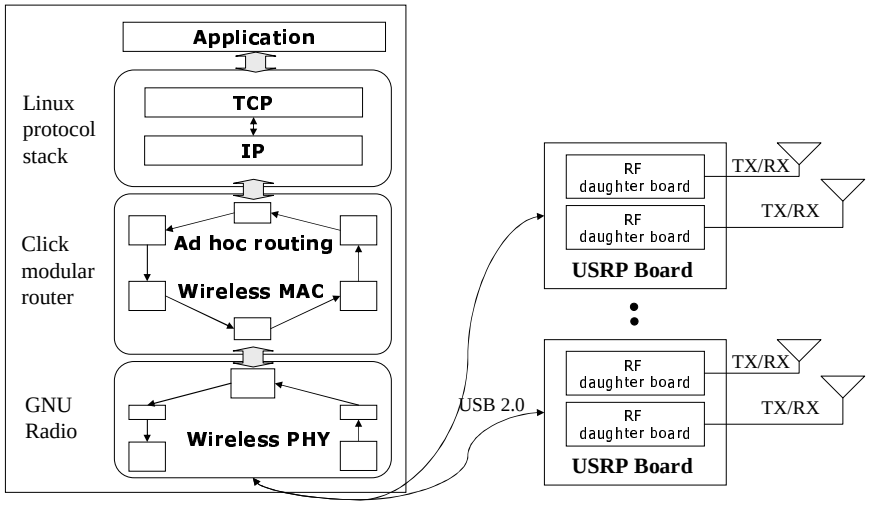


Figure 4.1: Hydra, a multihop wireless network testbed

Flexible Hardware

The USRP board is a flexible RF front-end that allows a Hydra node to communicate with other Hydra nodes. It is composed of four analog-to-digital converters, four digital-to-analog converters, a field programmable gate array, and two RF daughterboards. We can switch the RF daughterboards each of which supports signal transmission and reception over a different RF frequency band. Thus Hydra can support a wide range of frequency bands. Further, by connecting multiple USRP boards, Hydra can support the multiradio technology in which a node communicates with other nodes using multiple RF frequency bands simultaneously. The USRP boards are connected to the host system over a USB connection and are fundamentally controlled by the PHY on the host system.

Software Protocols

To allow the rapid prototyping of a variety of wireless protocols, Hydra implements all the network protocols as software running on the Linux operating system using the general purpose processors of the host system. Thus Hydra implements even the

PHY and MAC protocols as software. This is the key benefit of using Hydra since the PHY and MAC are normally implemented as unchangeable hardware logic in commercial off the shelf (COTS) products and thus it is not possible to experiment with new wireless PHY and MAC protocols using COTS wireless systems.

The PHY is implemented using GNU Radio [47], a software framework that allows us to implement a wide variety of signal processing algorithms running on a general purpose processor. Using the GNU Radio framework, we create a set of "signal processing blocks" each of which implements a signal processing algorithm and then compose a PHY protocol by connecting these small blocks into a signal processing flow graph. Thus we can easily implement a new PHY protocol by flexibly changing the connection graph and by reusing preexisting signal processing blocks. In Hydra, the PHY protocol is run in its own address space and communicates with the MAC running on another address space over a local UDP/IP connection that allows flexible interprocess communications.

The MAC is implemented using the Click modular router [33], a software framework that allows us to implement flexible and high performance network routers. The high level approach is almost the same as with GNU Radio. Using the Click modular router framework, we create a set of "packet processing elements" each of which implements some task required for packet processing such as packet classification and scheduling, and then compose a protocol by connecting these small elements into a packet processing flow graph. Thus we can flexibly configure a protocol by changing the connection graph and by reusing elements.

In Hydra, in addition to the MAC, the wireless ad-hoc routing protocols that are responsible for managing the connectivity of wireless networks are implemented using Click. Thus the MAC and the ad hoc routing protocols are run together in their own address space and are connected to the TCP/IP protocol stack that is run in the Linux kernel address space over a tunneling channel. This interface allows the

TCP/IP stack to exchange packets with Click and thus for a network application that is built upon the TCP/IP stack to send or receive the packets by using the wireless protocols provided by Hydra. This allows end-to-end experiments that test new wireless protocols using a wide variety of network applications such as web servers and browers, ftp, and ping that have been already implemented using the TCP/IP stack.

The key aspects of Hydra as an implementation context of our concrete architecture is that the software protocols of Hydra are composed of three different software frameworks each of which is run in its own address space and each of which has its own style of protocol implementation. Thus overall Hydra forms a layered architecture that connects each protocol module to one another using flexible interfaces.

Hydra is currently operational[1]. The PHY implementation is similar to the IEEE 802.11a standard [48], which supports orthogonal frequency division multiplexing (OFDM) technology [49] with multiple transmission rates. The MAC is essentially the 802.11 DCF MAC protocol [31] we have discussed to explain the rate control example in Section 2.1.1. The ad hoc routing protocols that have been already implemented in the Click modular router by other researchers [50, 51] are used for experiments. We have tested the Hydra implementation by using a set of network applications such as ping and ftp that exchange packets with another Hydra node over a multihop path.

4.1.2 Abstractions of Protocol Modules

Fig. 4.2 shows the three software frameworks that are used to implement the software protocols of Hydra and presents how we can generalize these different protocol

[1]I was involved in all the key phases in developing Hydra. I designed an implementation structure of the MAC, and then implemented the MAC using the Click modular router. Further I was involved in designing an implementation structure of the PHY using GNU Radio. This allowed me to easily deal with Hydra as a target system of our architecture.

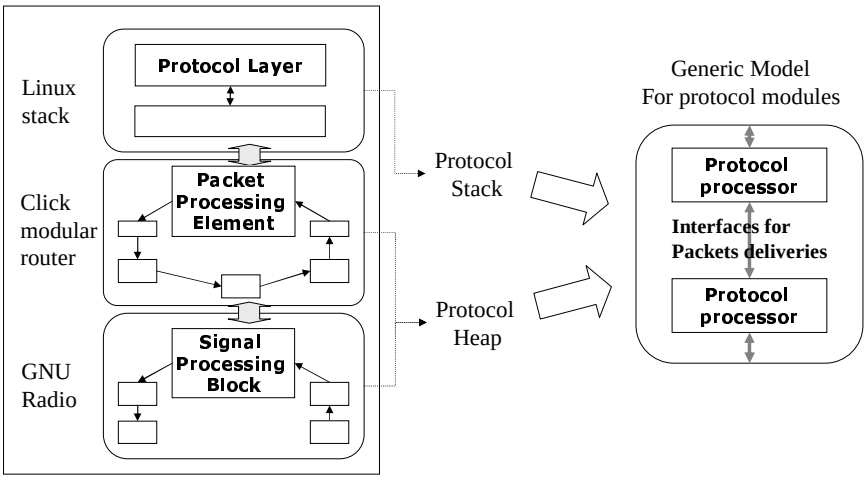


Figure 4.2: Software protocols of Hydra and their abstraction

modules into a generic model that describes the key aspects of the modules. This generic model allows our architecture to deal with a variety of protocol modules by capturing their key aspects.

As an intermediate stage of abstraction, we first classify these three protocol modules into two types. Based on the way protocols are implemented, we classify the protocol modules into a layer based "protocol stack" [52] or component based "protocol heap" [53]. The Linux protocol stack is a layer based "protocol stack" that follows the fundamental structure of the layered architecture. TCP at the Transport layer and IP at the Network layer implement their own tasks while hiding the details from other layers and then exchange packets with other layers using simple interfaces.

In contrast, GNU Radio and the Click modular router are component based "protocol heaps" in which the primary goal is to increase the level of flexibility in implementing protocols. A set of protocol modules [34, 54, 35, 55, 56] that are proposed for flexible and configurable protocol implementations can also be classified into this type. This type of protocol module divides the task of a protocol

into a set of smaller components such as the signal processing blocks of the GNU Radio and then composes a protocol by connecting them into a configurable graph. Thus they allow flexible and extensible implementation of a new protocol by reusing components and by reconfiguring the connection graph. The connection graph can allow a component to connect to any other component and thus does not introduce a strict layering boundary. However a component is implemented by hiding its details from other components and communicates with other components using a limited number of interfaces mainly for packet delivery. Thus as for the protocol stack case, the protocol heap still requires enhanced interactions between the components to support a variety of cross-layer adaptation.

As the final stage of abstraction, we generalize the two types of protocol modules into a generic model. The two key components in the generic model are the "protocol processor" and the "interface" for packet delivery. The "protocol processor" is a base unit of modular implementation of a protocol and thus generalizes the protocol layer of the protocol stack and a component in the protocol heap. Then the "interface" connects a protocol processor to another and allows the protocol processors to exchange protocol packets. Thus based on the generic model of protocol modules, our architecture inserts Interceptors between any two protocol processors by intercepting the packet delivery interface and attaches Adaptors to protocol processors to augment interfaces of the protocol processors.

4.2 A Concrete Architecture for Hydra

We now create a concrete architecture based on Hydra. The key challenge is that Hydra is composed of three different protocol modules each of which is running in its own address space and each with its own style of implementing a protocol. Thus we first show how our concrete architecture extends our conceptual architecture to coordinate the multiple protocol modules of a Hydra node. Then we present

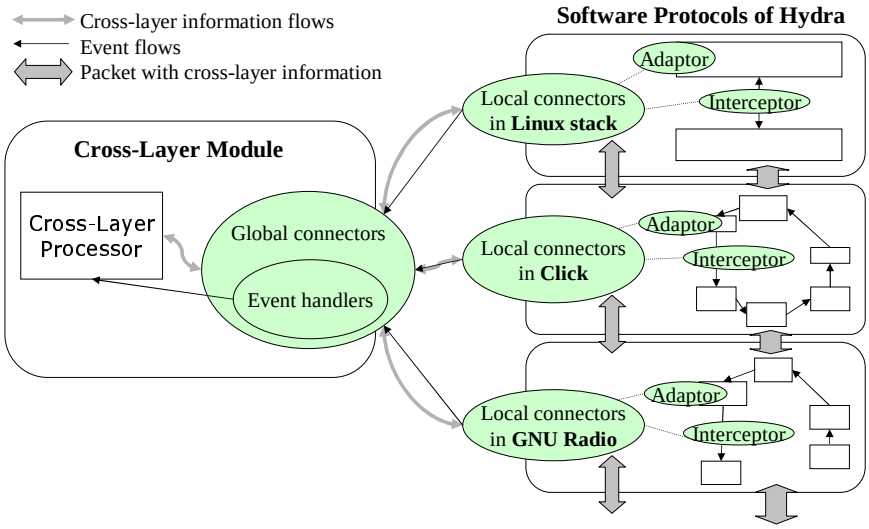


Figure 4.3: The overall structure of our concrete architecture designed for Hydra system

the detailed issues that can arise when we implement individual components of our concrete architecture on Hydra by grouping them into data, connecting, and processing components.

Finally, in the interest of performance, we further refine the proposed concrete architecture and show another way of implementing our architectural framework. This new concrete architecture aims to improve performance by reducing the overhead that can occur when we allow the communications between protocol processors and cross-layer adaptations without significant changes to the existing protocol implementation.

4.2.1 The Key Extensions

Fig. 4.3 shows the overall structure of our concrete architecture for Hydra and shows how we extend our conceptual architecture to coordinate the multiple protocol modules in a single Hydra node. We observe two problems in allowing our conceptual architecture to coordinate multiple protocol modules. First, each protocol module

is run in its own address space independent of the other modules. Second, the way of implementing protocol processors is different for each protocol module. GNU Radio implements signal processing blocks while Click implements packet processing elements. Further the TCP/IP stack implements a protocol layer as a protocol processor. This means that the Adaptors and the Interceptors need to be implemented differently to conform to each implementation environment. These make it hard to manage all three protocol modules by using the "Global connectors" provided by our conceptual architecture.

The key extension to address this problem is that all of the connectors are divided into two levels by introducing a set of "Local connectors". Each Local connector is implemented conforming to the protocol implementation environment provided by each protocol module and run in the address space of each protocol module. Thus the Local connectors can easily manage the Adaptors and Interceptors that are implemented using the same implementation environment and run on the same address space. Further the Local connectors allow a node to exchange cross-layer information with other network nodes using the out-of-band delivery path. The Local connectors create a packet that carries cross-layer information and then transmits and receives the packet by creating a new packet delivery path in the protocol modules. The Local connectors then communicate with the Global connectors, which coordinate the multiple Local connectors and interface with the cross-layer processors. This allows us to achieve cross-layer adaptations independent of the infrastructures and to freely change its operation without significant impact on the existing protocol modules.

4.2.2 Detailed Implementation Issues

We discuss the detailed issues that arise when we implement our architectural framework in a real wireless system. As we will show in later chapters, we implemented

the concrete architecture within Hydra. The primary goal of the implementation was to show how our architectural framework can be implemented in a real wireless system. Thus our implementation considered the key mechanisms provided by our architecture which can support the implementations of a wide variety of cross-layer adaptations, although we did not attempt to optimize performance in a significant way. We present the issues that occurred when we implemented the concrete architecture by grouping the architectural components into data, connecting, and processing components as previously.

Data Component

The first architectural component we will discuss the implementation issues of is the data component. In our architecture, cross-layer data can frequently be moved across address spaces by the connectors. The In-band connector moves cross-layer data from one protocol module to another, and the Out-of-band connector delivers data from a protocol module to a cross-layer module. Further Internode connectors can send the data to another node. Thus cross-layer data must frequently be marshalled and unmarshalled whenever a connector delivers the data to another address space. Thus we implemented a mechanism that can be used by the connectors to convert the data used in one address space into a message format that all the connectors agree upon. In our implementation, we defined a simple message format that is composed of the three key attributes of cross-layer data such as name, length and value. We then implemented a mechanism that converts the attributes into a single string message.

Another possible interesting solution is to use a standardized data representation such as extensible markup language (XML) [57]. However implementing the standardized representation would significantly complicate the implementation of our architectural framework. Further, such a general solution would not be useful

for our simple proof of concept. Thus the implementation of the data component using the standardized data representation was not pursued.

Connecting Components

In our concrete architecture, the Global connectors allow cross-layer processors to communicate with the protocol modules by fundamentally providing two key mechanisms, one for data delivery, and the other for event notification. Thus the data connectors provide interfaces with which a cross-layer processor requests that a data connector push or pull data. The event handlers provide an interface with which an event handler notifies the cross-layer processor of a certain event. For example, after the Synchronous event handler notifies the rate control processor of a packet passage event, the rate control processor requests the channel status to the Intranode version of the Out-of band connector.

However, the implementation of the interfaces is similar for both data connectors and event handlers. For example, the Intranode version of the In-band connector required the name of an Interceptor which piggybacks cross-layer data on a packet while the Synchronous event handler required the name of the Interceptor whose packet passage event a cross-layer processor listens to. Similarly, the Intranode version of the Out-of-band connector and the Asynchronous event handler required the name of the Adaptor with which a cross-layer processor communicates. Thus we defined a simple message format that is composed the three key attributes common for data delivery and event notification, the name of the data connector or event handler, the name of protocol module, and the name of the Adaptor or the name of the Interceptor. We then extend the message format for data delivery to designate the name of data that an adaptation process pushes or pulls. This message format is further extended for event notification to carry the name of the event by which the event handlers identify an event and then notifies an appropriate

cross-layer processor. For example, when an Interceptor in Click receives a CTS, the Synchronous event handler notifies the rate control process of a "CTS reception" event. Then the rate control process requests that the Out-of-band connector pull the "channel status".

The Internode version of the In-band connector uses the same message format as the Intranode version does, since both the In-band connectors communicate with the Interceptors. Further the Internode node version of the Out-of-band connector required a simple change in the message format. Since the Internode version of the Out-of-band connector allows the cross-layer processor to communicate with another cross-layer processor in other nodes, it required the network address of a node and the name of cross-layer processor with which the cross-layer process communicates, instead of the name of the Adaptor or Interceptor.

Processing Component

Implementing a cross-layer processor is not affected by the changes made by our concrete architecture, since the Global connectors hide the details of the underlying system implementations from cross-layer processors. Further in our implementation, the Global connectors provided uniform interfaces that allow cross-layer processor to communicate with the protocol modules and other nodes using a simple message format. Such a platform-independent implementation environment allows an adaptation process to be implemented as flexible and extensible a manner as possible and further one implementation of a cross-layer adaptation can be run on another wireless system that implements our architectural framework.

One interesting solution to support such an implementation environment is to connect our architectural infrastructure to a high-level programming language such as Python [58] or TCL [59]. By simply wrapping the interfaces provided by the Global connectors, we can implement a cross-layer processor using such a flex-

ible and platform-independent programming language. In our implementation, our architectural framework is written in the C++ programming language. Thus using the simplified wrapper and interface generator (SWIG) [60], which is a "software development tool that connects programs written in C and C++ with a variety of high-level programming languages", can allow us to easily extend our Global connectors to support flexible and platform-independent implementation environments. However this would complicate our simple proof of concept and thus using a platform-independent programming language was not pursued.

4.2.3 Further Refinements

Thus far, we have proposed a concrete architecture that implements our architectural framework while maintaining all the goals of our conceptual architecture. Thus the proposed concrete architecture allows us to implement a variety of cross-layer adaptations while hiding the details of the underlying protocol implementations. However, a key problem of the proposed architecture is the additional processing overhead to allow the communication between protocol processors and cross-layer adaptations. In our implementation of the concrete architecture, protocol processors and cross-layer adaptations run in their own address spaces and thus require interprocess communication over the Global and Local connectors. In the interest of performance, we propose another concrete architecture that can reduce the communication overhead of the proposed concrete architecture while maintaining to a significant degree the modularity of existing protocol implementations.

Overheads of Concrete Architecture for Hydra

To refine our approach with respect to performance, we first analyze possible overheads of our concrete architecture. Fig. 4.4 shows a concrete architecture in which a cross-layer processor communicates with the Adaptors and Interceptors over the

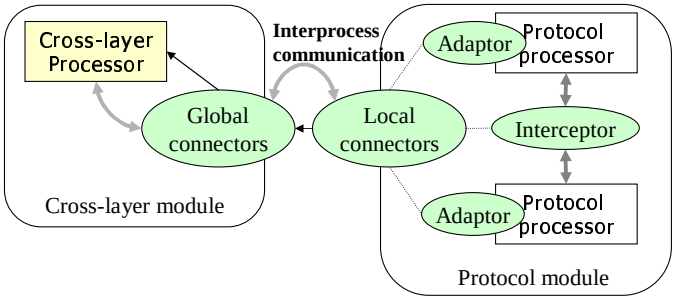


Figure 4.4: Communications between cross-layer processor and protocol processors in our concrete architecture

Global and Local connectors. This approach allows cross-layer adaptations to communicate with the protocol modules while hiding the detailed implementations of the protocol module.

However, this implementation approach introduces several processing overheads. First, the Global and Local connectors introduce additional processing overhead to allow the communication between cross-layer processors and the protocol modules. When a cross-layer processor requests that a Global connector push or pull data, the Global connector routes the data request to the Local connectors in an appropriate protocol module. Then the Local connectors deliver the request to an Adaptor or Interceptor. For example, to allow the rate control process to push the selected rate to the PHY, the Intranode version of the Out-of-band connector delivers the rate to the Local connectors in the GNU Radio and then the Local connectors set the rate by communicating with an Adaptor. Similarly to allow the cross-layer processor to be notified of an event, our concrete architecture also requires the mediation of the Global and Local connectors.

Secondly, the Global connectors and the Local connectors run in different address spaces and thus communication between them requires interprocess communication. In our implementation of the concrete architecture, the Global connectors communicate with the Local connectors by using UDP/IP sockets allowing flexible

communication using the loopback device. This interprocess communication introduces a context switch between the cross-layer module and the protocol module. Further it requires exchanging messages after marshalling and unmarshalling the cross-layer data.

Finally, an Interceptor can introduce another packet delivery step between two adjacent protocol processors, and an Adaptor can introduce an additional procedure to access information within a protocol processor.

Refined Concrete Architecture

To gain insight on how we can refine our approach to reduce overheads while preserving the modularity of existing protocol implementations, we measured the communication overhead of our architecture. As we will show in detail in later chapters, the measurements showed that most of overheads come from interprocess communication between cross-layer adaptation and the protocol modules, while processing overheads caused by the Global and Local connectors, Interceptors and Adaptors were insignificant. Thus the key refinement is to move the cross-layer processor to the address space of each protocol module. This refinement allows a cross-layer process and the Local connectors to run in the same address space as a protocol module and thus eliminates the interprocess communication between them. For example, we can implement the rate control processor to run in the address space of the MAC. Then the cross-layer processor communicates with the Interceptors and the Adaptors in the MAC without needing interprocess communication.

Fig. 4.5 shows how the previously proposed concrete architecture can be refined to reduce the communication overhead. This approach requires further changes to the cross-layer processor and the Local connectors. In the refined architecture, a cross-layer processor is implemented conforming to the implementation environment provided by a protocol module. Then the Local connectors are extended to allow

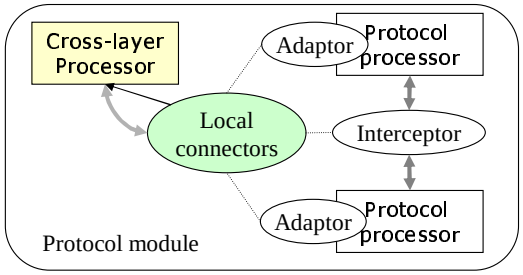


Figure 4.5: Communications between cross-layer processor and protocol processors in refined architecture

the cross-layer processor to communicate with the Interceptors and Adaptors without interprocess communication. However, as we will show, the refined architecture allows the modularity of both cross-layer adaptations and the existing protocol implementations. In our implementation, the Local connectors allowed a cross-layer processor to communicate with the Interceptors and Adaptors using the same mechanism that was provided by the Global connectors. Further the existing Interceptors and Adaptors were used without modification. Thus the refinement shows another concrete architecture that can support systematic and modular implementation of cross-layer adaptations without significant performance degradation.

4.3 Existing Cross-layer Architectures

Our concrete architecture has shown how we can extend our conceptual architecture to implement our architectural framework within Hydra. Thus this validates our conceptual architecture as a generic one that can derive a cross-layer architecture for a particular wireless system. We further validate how our conceptual architecture serves as a generic architecture by presenting some preexisting cross-layer architectures and analyzing their mechanisms in terms of our architecture.

We present two cross-layer architectures, the "efficient cross-layer architecture for wireless protocol stacks" (ECLAIR) [14] and the "mobile metropolitan ad-

hoc networks" (MobileMAN) system [13]. To our best knowledge, these are the most sophisticated cross-layer architectures among the ones that have been proposed in the literature [16, 40, 41, 42, 43, 44, 45, 15] to coordinate the adaptation processes with existing protocol implementations. These two architectures aim to provide a systematic framework that can be used to implement a wide variety of cross-layer adaptations. However, they provide only some of the mechanisms among the ones that our architecture provides. Thus we can describe these architectures as possible instantiations of our architecture and validate how our architecture serves as a generic model that describes a wide variety of cross-layer architectures.

ECLAIR can represent a set of existing architectures [16, 40], which implement the adaptation processes outside of the protocol module to coordinate the implementations with the existing protocols. MobileMAN shows a typical model of a set of other existing architectures [41, 42, 43, 44, 45, 15], which maintain the modularity of existing protocol implementations by allowing a protocol layer to indirectly communicate with another layer over a Global connector. Thus describing the two architectures allows us to show that our architecture covers a wide variety of preexisting cross-layer architectures and thus can serve as a generic model.

We first illustrate the basic operations of the individual cross-layer architectures and analyze the mechanisms they support using our architecture.

4.3.1 ECLAIR

The first cross-layer architecture we present is ECLAIR [14]. The main goals of ECLAIR are similar to those of our architecture. ECLAIR aims to achieve rapid and portable implementations of cross-layer adaptations while maintaining the advantages of modularity found in the existing "protocol stack". Thus, as in our architecture, ECLAIR implements cross-layer adaptations outside the protocol stack and allows adaptations to access information inside of the protocol layer without

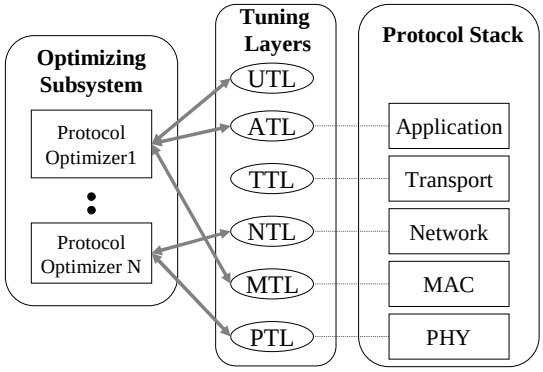


Figure 4.6: The key components and their relationships in efficient cross-layer architecture for wireless protocol stacks (ECLAIR)

significant changes to the existing protocol implementations.

Fig. 4.6 shows the key components and their relationships in ECLAIR. A tuning layer (TL) attached to a protocol layer provides interfaces which allow protocol optimizers (POs) to read and update data inside the protocol layer and also to be notified of a certain event generated from the protocol layer. Then a PO implements a cross-layer adaptation outside of the protocol stack by using TLs. For example, in ECLAIR, a PO that implements the rate control process can use TLs attached to the MAC (MTL) and the PHY (PTL) to obtain packet length and channel status information.

The TLs in ECLAIR can be viewed as the Intranode version of the Out-of-band connector and the Asynchronous event handler that allow an adaptation process outside of the protocol module to access the existing protocol processors. However, ECLAIR does not consider the in-band and the internode delivery cases. Further ECLAIR does not support the Synchronous event handler, since it takes the view that a synchronous adaptation process can block the operations of a protocol processor and can introduce a performance degradation to the existing protocol process. However, our taxonomy has shown that there are cases where the adaptation process is required to be synchronized with packet processing. For example,

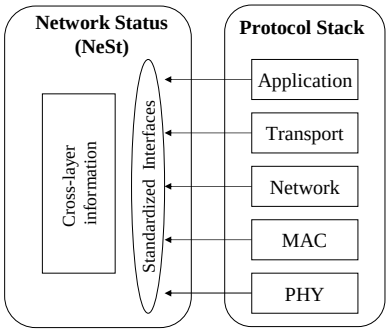


Figure 4.7: The key components and their relationships in the mobile metropolitan ad-hoc networks (MobileMAN) system

the rate control process selects an appropriate rate for every packet transmission. As we will show in Section 6, we implemented rate control within Hydra. Performance measurements of the implementation showed that optimizing the connectors can to a significant degree reduce the additional processing time that the protocol modules spend executing the synchronous process, and minimize the performance degradation.

4.3.2 MobileMAN

The second cross-layer architecture is the MobileMAN system [13]. The main goal of MobileMAN is to provide standardized interfaces that can be used to implement cross-layer adaptations while preserving the modularity of the existing protocol implementation. However MobileMAN disregards the advantages of modular design and implementation of cross-layer adaptations and thus merges the adaptation processes into the protocol processors.

Fig. 4.7 shows the key components and operations of MobileMAN. In MobileMAN, an adaptation process is implemented inside of the protocol processor and thus a protocol may directly communicate with another one. Thus MobileMAN introduces the Network Status (NeSt) which is a Global connector that mediates all

the cross-layer interactions including data exchanges and event notifications. The NeSt provides standardized interfaces that allow a cross-layer protocol to communicate with another protocol indirectly through the NeSt. Thus the NeSt allows a protocol layer to be maintained or replaced with a new release without affecting other protocols. However, the changes to the cross-layer adaptation process can cause modification or replacement of existing protocol implementations.

The NeSt can be viewed as the Intranode version of the Out-of-band connector and that of the Asynchronous/Synchronous event handler. Using the NeSt, a protocol layer delivers data to another protocol by using a path that is not related to packet delivery. Further a protocol layer notifies another protocol of any events that occur inside that protocol layer. For example, the rate control process that is implemented inside the MAC can notify the NeSt (and thus the PHY) of a Synchronous event when the MAC receives a CTS packet. Further the protocol reconfiguration process inside the PHY can notify the NeSt of an Asynchronous event as soon as it detects a new standard. However, MobileMAN does not pay much attention to the in-band and the internode delivery cases, since its main concern is the indirect communications between layers within a node over the Intranode version of connector.

Chapter 5

Evaluation Overview

The goal for our architecture is to provide the key mechanisms that are required to implement the wide variety of cross-layer adaptations described by our taxonomy. Further our architecture is designed to maintain to a significant degree the advantages of modularity in existing protocol implementations by allowing cross-layer adaptations to be implemented outside of the protocol modules. Thus our claim is that our architecture supports the implementation of a wide variety of cross-layer adaptations reducing to a significant degree changes to the existing protocol implementations.

To validate this claim, we performed a series of case studies. Each case study implements a cross-layer adaptation within Hydra based on our concrete architecture. As case studies, we implemented three cross-layer adaptations, and each case study is presented as an independent chapter. The first goal of the case studies is to implement and evaluate all the key mechanisms provided by our architecture. We selected the three case studies each of which requires a different set of mechanisms for information delivery and event notification. Thus, implementing the three adaptations allowed us to implement and evaluate all the key components of our architecture. The second goal is to show that our architecture allows us to easily

implement a variety of cross-layer adaptations while preserving the modularity of existing protocol implementations. Thus, in addition to implementing the proposed adaptations based on our framework, we also implemented the same adaptations in a "conventional" way in which software protocols in Hydra directly communicate with each other. Comparing both implementation approaches allows us to evaluate how effectively our architecture supports the implementation of cross-layer adaptations. The final goal is to evaluate the performance of our architecture. Our performance measurements showed the mechanisms that introduce overhead in our architecture and thus allowed us to find a way of refining our approach. We implemented the adaptations based upon the refined architecture and evaluated how our architecture supports the implementation of the adaptations without significant performance degradation.

We begin by presenting a set of high-level goals of the case studies. We then show how we used the three cross-layer adaptations to implement and evaluate our architecture, followed by a discussion on how we present the implementation and evaluation results.

5.1 Goals of the Case Studies

To show an overview of our evaluation using the case studies, we present three high-level goals of the case studies. Table 5.1 shows the three cross-layer adaptations we used and presents the three goals. Our first goal is to implement and evaluate all the key mechanisms provided by our architecture. To achieve this goal, we used three cross-layer adaptations, which are rate control, contention window control, and a link-aware routing protocol, as labeled in the first row of the table. We selected the three adaptations, each of which requires a fundamentally different set of mechanisms for information delivery and event notification. Thus implementing all three adaptations allow us to test all the key components of our architecture.

Table 5.1: Three high-level goals and detailed evaluation items for the case studies

Evaluation Items \ Cross-layer adaptations	Rate Control	Contention Window Control	Link-aware Routing Protocol
1. The first goal: - To implement and evaluate all the key mechanisms of our architecture			
1.1 Data connectors			
- Intranode /In-band data connector	O		
- Intranode /Out-of-band data connector	O	O*	O*
- Internode /In-band data connector			
Path control / One-hop case	O		O*
Path control / Multi-hop case			
- Internode /Out-of-band data connector			
Area control / One-hop case		O	O*
Area control / Multi-hop case			O
1.2 Event Handlers			
- Synchronous event handler	O		O*
- Asynchronous event handlers		O	O*
2. The second goal: - To evaluate how our architecture supports the implementation and extension of adaptations			
2.1 Modularity of protocol module			
- Minimum changes to the existing protocol implementations	O	O	O
2.2 Modularity of cross-layer module			
- Protocol module independent implementation	O	O	O
- Easy extension of adaptation process	O		O
2.3 Coordination of all modular components			
- Support an adaptation that communicates with multiple protocol layers using a set of architectural components			O
3. The third goal: - To evaluate how our architecture supports the implementations without significant performance degradation			
3.1 Reducing the communication overheads			
- Elimination of inter-process communication	O	O	

*: Evaluation by reusing the component.

Our second goal is to evaluate how our architecture supports the implementation and extension of a variety of cross-layer adaptations while maintaining modularity of the existing protocols. This is key, since it supports our claim that our architecture allows flexible and extensible implementation of adaptations while minimizing the modification of existing protocol implementations.

Our final goal is to evaluate the performance of our architecture. Thus we measure the performance of our architecture by using the implementation of our architecture within Hydra and show the overhead of our architecture. As we will show, our performance measurements showed that most of overhead comes from interprocess communication that occurs to allow an adaptation outside protocol module to communicate with existing protocol modules. This measurements allowed us to find a way of refining our architecture to reduce the communication overhead while maintaining the modularity of both the adaptation and the existing protocol implementation. By measuring the performance of our refined architecture, we evaluate how effectively our architecture supports the implementation of adaptations without significant performance degradation.

To achieve these later two goals, in addition to implementing the adaptations based on our architecture, we implemented the same adaptations in a conventional way. Comparing both implementation techniques allowed us to evaluate how easily we can implement adaptations. Further implementing the adaptations using our refined concrete architecture allowed us to evaluate how our architecture can minimize communication overhead and thus how well our architecture meets its goals without significant performance degradation.

5.2 Detailed Evaluation Items

To show how we used the cross-layer adaptations to achieve our goals, we decompose the three high-level goals into a set of detailed evaluation items. As further shown in

Table 5.1, the evaluation items for our first goal concern the individual components of our architecture such as data connectors and event handlers. For example, as shown by the evaluation item labeled as "Event Handlers", we implemented and evaluated the Synchronous event handler when we implemented rate control, while we implemented the Asynchronous event handler when we implemented contention window control[1]. Then, we re-evaluated the implemented event handlers by reusing them when we implemented a link-aware routing protocol.

In the three case studies, we were able to implement all the key mechanisms. The exception was the multihop version of path control mechanism that is provided by the Internode version of the In-band connector. The mechanism allows an adaptation to deliver information to a specific destination node that can be reached by multiple intermediate hops. Thus, in our concrete architecture, the mechanism is fundamentally the same with the one-hop version of path control mechanism since both the mechanisms piggyback information on a packet by communicating with the Interceptor. The difference is the destination of the packet on which the Interceptor piggybacks information. We will discuss the detailed operation of each adaptation in the following Chapters and show how the three case studies allowed us to implement and evaluate all the key mechanisms shown in Table 5.1.

The evaluation items for our second goal concern how our architecture supports the implementation and extension of cross-layer adaptations while maintaining the modularity of the existing protocol implementation. Using the three adaptations, we evaluated how our architecture reduces changes to the existing protocol implementations and compared the results with the conventional approach. Further we evaluated how our architecture allows the flexible and extensible implementa-

[1]Our second study implements contention window control instead of protocol reconfiguration that was discussed in Section 3.3.2. We selected contention window control since it allows us to implement and evaluate the mechanisms that protocol reconfiguration introduced, reducing the work required to implement two different MAC protocols. Further contention window control shows another cross-layer adaptation that uses the mechanisms provided by our architecture.

tion of each adaptation. For example, in our first case study, we show how we implemented rate control based on our platform-independent implementation environment. Then we present how we were able to extend the implementation to another version of rate control. Further we evaluated how our architecture coordinates complex cross-layer interactions. For this evaluation item, we used the link-aware routing protocol which required complex interactions across the multiple layers and also across network nodes. This case study also shows how our architecture allows the implementation of the new routing protocol using the existing cross-layer rate control adaptation.

The evaluation items for our final goal concern the performance of our architecture. We show how our architecture reduced the communication overhead using the implementation mechanisms of our refined concrete architecture. For example, in our first case study, we inserted rate control inside the address space of the MAC. This refinement allowed rate control and the MAC to run in the same address space and thus eliminated the interprocess communication between them. Since most of overhead of our architecture came from the interprocess communication required for the communications between adaptation and the existing protocols, the refinement significantly reduced the overhead. This implementation technique introduced further changes in the adaptation conforming to the implementation environment provided by protocol modules. However, our framework components such as Local connectors, Interceptors and Adaptors allowed the implementation of the adaptation to be loosely coupled with the existing protocol implementations. Using rate control and contention window control, we show how our architecture reduced the overhead while maintaining to a significant degree the modularity of both the protocol layers and the adaptation. However, the performance evaluation using link-aware routing protocol is not performed, since we implement this adaptation using the mechanisms implemented and evaluated by rate control and contention window control.

5.3 Metrics for Evaluations

Fundamentally our evaluation is based on analyzing the experience obtained while implementing the adaptations. In each case study, we show the detail of the implementations both in a conventional approach and based on our concrete architecture. We then show the evaluation results by presenting comparisons between both implementation techniques. This comparative analysis allowed us to qualitatively evaluate how effectively our architecture supports a variety of cross-layer adaptations. Further, to show a quantitative comparison of both implementation techniques, we developed metrics that can be used to evaluate our architecture as well as a conventional implementation. These metrics allowed us to manifest the benefits and drawbacks of using our architectural framework.

To understand how we quantitatively evaluate the implementations of adaptations, we show the metrics that we used in our case studies. We first show metrics that evaluate how our architecture supports the implementation of adaptations maintaining modularity of the existing protocol implementations. We then show metrics that measure the performance of our architecture.

5.3.1 Measuring Modularity

To quantitatively evaluate how our architecture supports the implementation of adaptations maintaining modularity of the existing protocols, we developed two metrics as shown in Table 5.2. The first metric is the number of protocol processors that were created and modified to implement an adaptation. This metric allows us to measure how many changes the implementation of an adaptation introduces into the existing protocol implementations. In Hydra, the Network and the MAC layers are composed of a set of packet processing elements in Click while the PHY is composed of a set of signal processing blocks in GNU radio. Thus to implement an adaptation we created and modified a set of protocol processors in Click, GNU radio,

Table 5.2: The metrics for evaluating impact of the implementation of adaptation on modularity of existing protocols

Metric	Evaluation item
The number of protocol processors: - created - modified	Changes in the existing protocol implementations caused by the implementation of adaptation.
The number of protocol processors that are involved in delivering: - cross-layer information	Interdependencies between protocol processors caused by the implementation of adaptation.

or in both. We observed that both creation and modification introduced changes to the existing protocol implementations. However, the impact of creation and modification on the modularity of the existing protocol implementations were different. For example, in the conventional implementation of rate control, we created a packet processing element in Click to allow the MAC to calculate the rate using the channel status. Such creation introduced a new protocol processor that is dedicated to the adaptation. However, we modified a packet processing element to allow the MAC to piggyback the channel status on the CTS. Such modification required the existing protocol processor to perform an additional process for rate control and thus introduced change in the protocol processor coupled with rate control. Similarly, in our architecture, we create an Interceptor as a new protocol processor. Thus the Interceptor reduces to a significant degree changes in the existing protocol implementations. However augmenting interfaces by attaching an Adaptor does not eliminate changes to the protocol processor. For example, to allow an Adaptor to access cross-layer information within a protocol processor, some variables inside the protocol processor need to be exposed.

Our second metric is the number of protocol processors that are involved

Table 5.3: The metrics for evaluating performance of the implemenation of adaptation

Metric	Evaluation item
The ratio of Adaptation time over Protocol processing time ($R_{ADAPT} = \frac{T_{ADAPT}}{T_{PROT}}$)	The impact of adaptation overhead on the performance of Hydra node
The ratio of Communication time over Adaptation time ($R_{COMM} = \frac{T_{COMM}}{T_{ADAPT}}$)	The impact of the communication overhead on the performance of adaptation.

in delivering cross-layer information. This metric evaluates how the adaptation process communicates with protocol processors in Click and GNU Radio and thus measures interdependencies between the protocol processors which the implementation of adaptation introduces. For example, in the conventional implementation of rate control, a signal processing block in GNU Radio delivers the channel status to the rate control process in Click by piggybacking the information on the data packet. To allow such an in-band information delivery, we modified a set of protocol processors such that a protocol processor transforms the information when it forwards the information to another one. Thus the implementation required a set of changes in the existing protocol processors, which introduced a series of interdependencies between them.

5.3.2 Measuring Performance

To measure the performance of the conventional implementation and using our architecture, we developed two metrics as shown in Table 5.3. The first metric is the ratio of the adaptation time over the protocol processing time (R_{ADAPT}). The ratio shows the additional processing time which a Hydra node spends performing the adaptation process and thus evaluates the overall impact of adaptation overhead

on the performance of a Hydra node. For example, in rate control, we measured the adaptation time (T_{ADAPT}) which rate control spends executing the adaptation process. Then we measured the protocol processing time (T_{PROT}) which the MAC and PHY spends transmitting a data packet. Since rate control executes a synchronous process that performs adaptation for every packet transmission, the ratio shows the additional processing time that occurs when a Hydra node transmits a data packet and thus evaluates the overall impact of adaptation on the performance of the Hydra system. However, an asynchronous process periodically performs adaptation regardless of packet processing time. Thus, in contention window control, which executes as an asynchronous process, we measured the period of the adaptation process to calculate the additional processing time a Hydra node spends performing the adaptation.

To measure the communication overhead of our architecture and compare it with that of the conventional approach, we developed another metric, the ratio of communication time over the adaptation time (R_{COMM}). This ratio shows how much time the adaptation process spends exchanging information and thus evaluates the impact of communication overhead on the performance of the adaptation process. For example, in rate control, we measured the communication time (T_{COMM}) which rate control spends exchanging the rate and channel status. We then measured the calculation time (T_{CALC}) which rate control spends calculating a rate using the channel status. The adaptation time (T_{ADAPT}) which rate control spends performing adaptation can be calculated by adding the two measurements. Since both the conventional and architecture-based implementations use the same algorithm for rate selection, the calculation time is same for both implementation techniques. Thus the adaptation time is determined by the communication time of each implementation technique.

Chapter 6

Case I: Rate Control

Our first case study is rate control. We have implemented both the Intranode and the Internode version of rate control as presented in Section 2.1. The first goal of this case study is to implement the Synchronous event handler to allow rate control to execute adaptation at the time of packet delivery. Further this case study requires that we implement the Intranode version of the Out-of-band delivery mechanism and the Intranode and Internode versions of the In-band delivery mechanisms. Fig. 6.1 shows our taxonomy and the mechanisms we have implemented for rate control. We first considered the Intranode version of rate control and thus implemented the Intranode version of the Out-of-band connector and the Synchronous event handler. We then extended this version of rate control to the Internode version and thus implemented the Internode version of the In-band connector. The second goal is to evaluate how easily we can implement rate control using our architectural framework. Thus we also implemented both the Intranode and Internode versions of rate control in a conventional way, in which the MAC performs the adaptation by directly communicating with the PHY. Then we compared both the implementation techniques. The final goal is to evaluate the performance of our architecture. Thus we measured the performance of our architecture. Our performance measurements

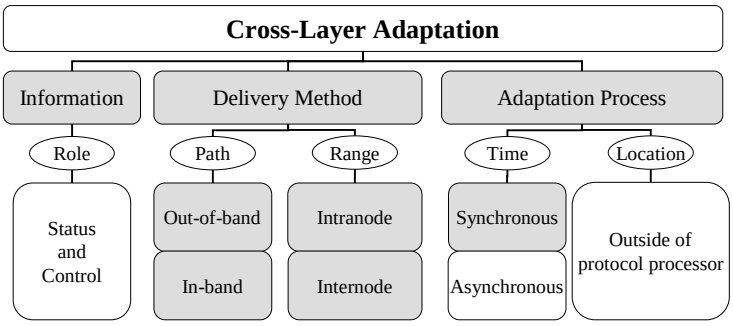


Figure 6.1: Mapping rate control to taxonomy

showed the mechanisms that introduce the overhead in our architecture and thus allowed us to find a way of refining our approach, with respect to what concrete architecture is desirable to support the implementation reducing the communication overhead of our architecture. We implemented the Internode version of rate control based on the refined architecture and then measured the performance of our architecture.

We begin by presenting the implementations of both the Intranode and Internode versions of rate control in a conventional approach, followed by showing the implementations based on our concrete architecture. We then show our evaluation by comparing both the implementation techniques. Finally we show how we refined our architecture to reduce the communication overhead and then present the performance of our architecture.

6.1 Implementations

To evaluate our architectural framework, we implemented rate control using our architecture as well as using the conventional approach. Further to evaluate how our architecture supports the implementation and extension of rate control, we implemented both the Intranode and the Internode versions of rate control. To show feasibility of the implementations, we briefly explain how we implemented

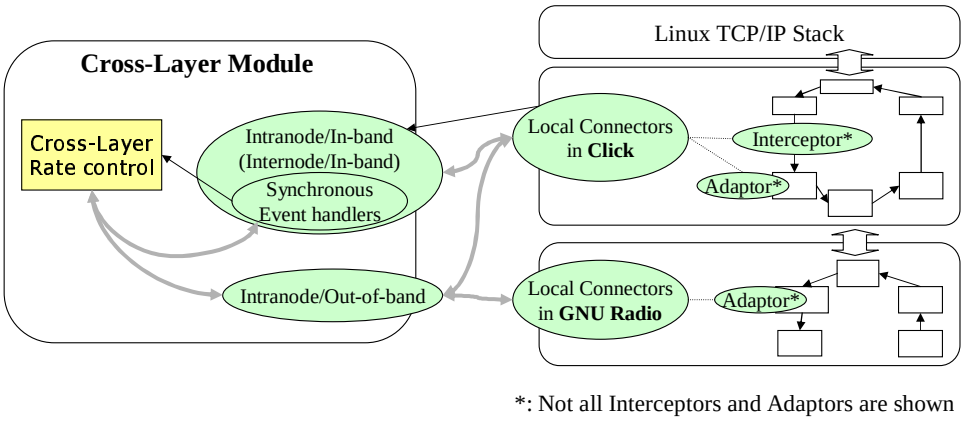


Figure 6.2: Implementation of cross-layer rate control by using our concrete architecture

rate control within Hydra. We then show how the implementations are working by presenting experimental results.

6.1.1 Implementation using Hydra

We implemented rate control using Hydra to show the feasibility of implementation based on our concrete architecture as well as in a conventional way. We first implemented the Intranode version of rate control in a conventional way. In Hydra, the MAC is composed of a set of packet processing elements in Click while the PHY is composed of a set of signal processing blocks in GNU Radio. To implement rate control, we created and modified a set of packet processing elements in Click and changed several signal processing blocks in GNU Radio. Then we changed the connection graphs of both Click and GNU Radio to compose the new MAC and the PHY protocols.

We then implemented the Internode version of rate control by extending this Intranode version. This extension required further modification of packet processing elements in Click and then required changes to the connection graph of Click. We present further detail of these implementations in Section 6.2.

We then implemented rate control based on our architectural framework. Fig. 6.2 shows how we implemented rate control using our concrete architecture. The key is that the Local connectors, Interceptors, and Adaptors were implemented as packet processing elements in Click and signal processing blocks in GNU Radio. To implement the Intranode version of rate control, we first implemented the Local connectors, Interceptors and Adaptors and inserted the architectural components into Click by changing the connection graph. Similarly for the PHY, we implemented and inserted the Local connectors and Adaptors into GNU Radio. The Intranode version of the In-band connector and its Synchronous event handler were implemented as Global connectors. These Global connectors communicate with the Local connectors in Click and GNU Radio by using UDP/IP sockets allowing flexible interprocess communication using the local loopback device. Similarly, the Intranode version of the Out-of-band connector was also implemented as a Global connector. Finally, we implemented the rate control processor that communicates with both the MAC and PHY using Global connectors.

We implemented the Internode version by extending the Intranode implementation. The Internode version of rate control required the Internode version of the In-band connector. In our implementation, the Intranode version of the In-band connector also handles the Internode version of the In-band delivery mechanism, because both delivery mechanisms access the existing packet by interacting with the Interceptor.

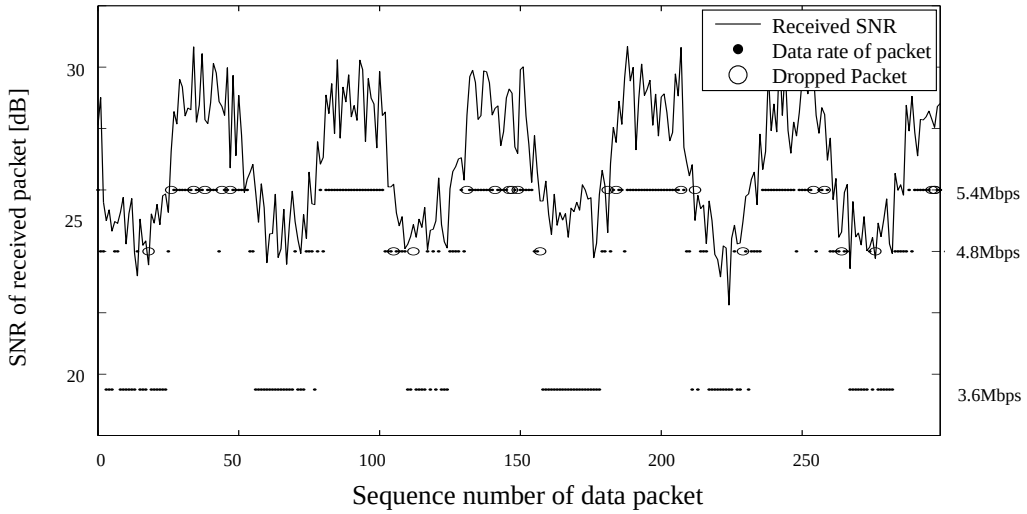
6.1.2 Validation of Implementation

To show that the implementations are operational within Hydra, we present some experimental results. We performed a set of experiments using several rate control protocols that we implemented within Hydra and presented the experimental results in [61]. The Internode version of rate control that we used for the experiments had

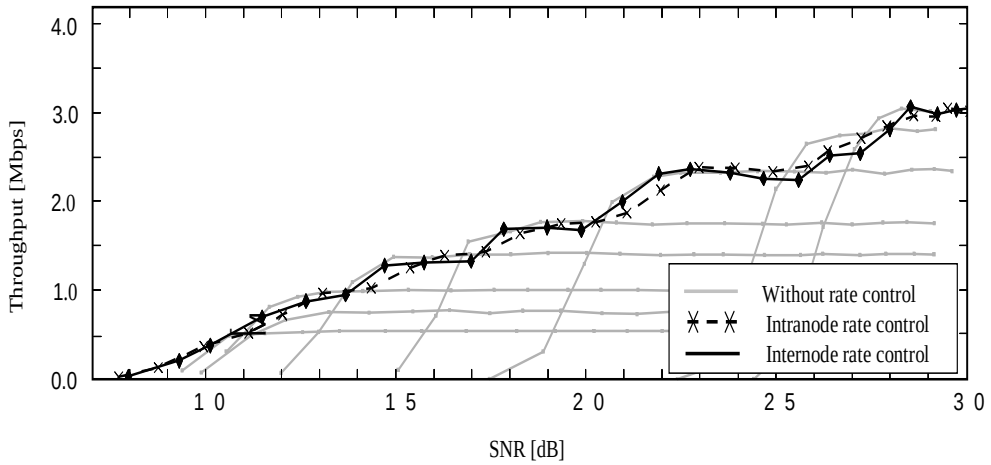
a few differences in detailed implementation from that of this case study. The main goal here is to show that the implementation of rate control is working within Hydra by presenting how rate control selects an appropriate rate using the channel status, rather than to explore and evaluate the performance of the algorithm. Thus we validate the implementations of rate control by presenting some experimental results that we presented in [61].

The Internode version of rate control that we used in these experiments was receiver based auto rate control (RBAR) [30]. The RBAR protocol has a few differences from the Internode version of rate control of this case study. First, in our Internode version of rate control, after a receiver measures channel status, it informs the transmitter of "channel status". Then the transmitter calculates the rate. In RBAR, however, after a receiver measures the channel status, it calculates the rate. Then it informs the transmitter of the selected "rate". Thus RBAR piggybacks the "rate" on the CTS while our Internode version of rate control piggybacks "channel status" on the CTS. However both rate control algorithms select an appropriate rate for every packet transmission using the channel status measured at the receiver.

Secondly, our Internode version of rate control accounts for the channel status and the length of the data packet to select a rate appropriate for each packet [23]. However, RBAR considers only the channel status for rate selection. To allow RBAR to select a rate based on the packet size, it requires a few changes in the MAC implementation. After the MAC a transmitter piggybacks the length of the data packet on the RTS, RBAR needs to obtain this information by accessing the RTS. Then it needs to calculate the rate accounting for the data packet size. However both our Internode version of rate control and RBAR select the rate for each data packet transmission using the same criterion, signal-to-noise ratio (SNR) as measured at a receiver. Thus the performance of both rate control algorithms will be fundamentally the same for a fixed size of data packet, as the experimental results show in Fig. 6.3.



(a) MAC-level trace of Internode version of rate control



(b) Throughput of Intranode and Internode rate control

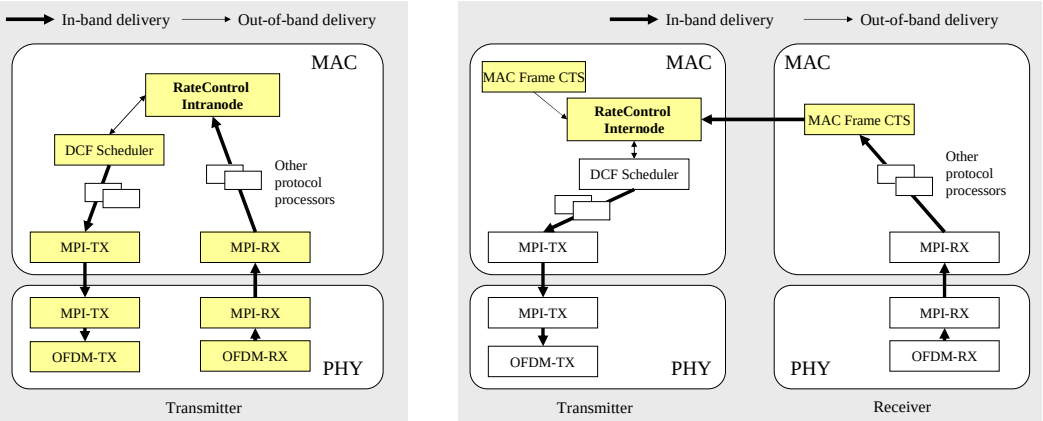
Figure 6.3: Experimental results of cross-layer rate control

Our experimental setup consisted of two Hydra nodes, one acting as a transmitter and the other as a receiver. We created a traffic flow in which the transmitter periodically sends a packet to the receiver executing the rate control process. We then measured the SNR, the rate and transmission error for each transmitted packet to trace the detailed behavior of rate control. Fig. 6.3(a) shows how the Internode version of rate control works when the wireless channel varies with time. The X-axis is the sequence number of the transmitting packet. The left Y-axis shows the SNR of the received packet, while the right Y-axis shows the rate selected for packet transmission. The line shows the received SNR for each packet and small dots show packets that were successfully received and larger open circle show packets that had errors. As expected, rate control tracks the channel well and generally selects an appropriate rate for each data transmission.

Fig. 6.3(b) shows the throughput of both the Intranode and Internode versions of rate control. The X-axis shows the SNR and the Y-axis shows the measured throughput. Light lines show the throughput achieved by the fixed rates supported by the PHY, and two dark lines show the throughput achieved by each rate control process. As further described in [62, 61], the results can vary with respect to how the criterion for rate transition is chosen and how fast the wireless channel varies. This result shows that, however, after choosing an appropriate criterion, both the rate control processes can increase the throughput of the wireless link by selecting an appropriate rate for each data transmission.

6.2 Evaluation

To evaluate how our architectural framework supports the implementation and extension of rate control, we compare the implementation based on our architecture with that in the conventional approach. To show our comparison, we first present detailed implementation and evaluation results for the conventional approach. We



(a) The Intranode version

(b) The Internode version

Figure 6.4: A conventional implementations of rate control

then show the detailed implementation and evaluation of our architecture. Finally, we show the comparative analysis of both the implementation techniques.

6.2.1 Conventional Implementation

Implementation showed that the conventional approach required a set of changes to the existing MAC and PHY implementations. Fig. 6.4 shows how we modified the packet processing elements in Click and signal processing blocks in GNU Radio. A difference of this conventional implementation from the operation shown in Section 2.1.1 was that the PHY informs the MAC of the channel status by piggybacking the information on every received packet as the MAC does to deliver the rate to PHY. Thus, instead of the MAC acquiring the channel status from the PHY by making a direct call, now the MAC accesses the channel status inside the packet's internal structure. We first implemented the Intranode version of rate control. As shown in Fig. 6.4(a), implementing the Intranode version of rate control required the following steps:

1. A packet processing element was created in Click:

    - to allow the MAC to calculate the rate using the channel status from the PHY and the length of the data packet.

2. A set of packet processing elements in Click were changed:

    - to allow the MAC to obtain the piggybacked channel status from packet delivered from the PHY, and
    - to allow the MAC and PHY to use the selected rate.

3. A set of processing blocks in GNU Radio were changed:

    - to allow the PHY to piggyback the channel status on a packet, and
    - to allow the MAC to change the rate.

4. The interface between the MAC and PHY were changed:

    - to allow the channel status and the rate to be marshalled and unmarshalled when they move between the MAC and the PHY.

The key problem was that these changes caused individual protocol processors to become interdependent. For example, in our initial implementation of rate control, the channel status information that the MAC and PHY exchanged was an integer valued received signal strength indication (RSSI) [31]. However, to allow rate control to have a better estimate of the channel status, we changed rate control to use a floating point valued signal to noise ratio (SNR). Thus we needed to change the MAC and PHY implementations to exchange the SNR instead of the RSSI. This required that the interface between the MAC and PHY and the protocol processors that deliver the channel status change to deal with the new type of channel information.

We then extended the Intranode version of rate control to the Internode version. This extension required modification of a few more packet processing elements

in Click. As shown in Fig. 6.4(b), implementing the Internode version of rate control required the following steps:

1. A set of packet processing elements in Click were changed:

    - to allow the MAC to use the new CTS packet format that delivers the channel status from the receiver to the transmitter, and
    - to allow the MAC to execute rate control using the channel status piggybacked on the new CTS packet.

The problem here was that the dependencies between the Click elements changed. Rate control now acquires the channel status that is piggybacked on the CTS packet, while the Intranode version acquired the channel status by accessing the Click's internal data structure that holds the channel status of the received packet. Thus the rate control process became dependent on a packet processing element that exchanges the channel status using the CTS packet.

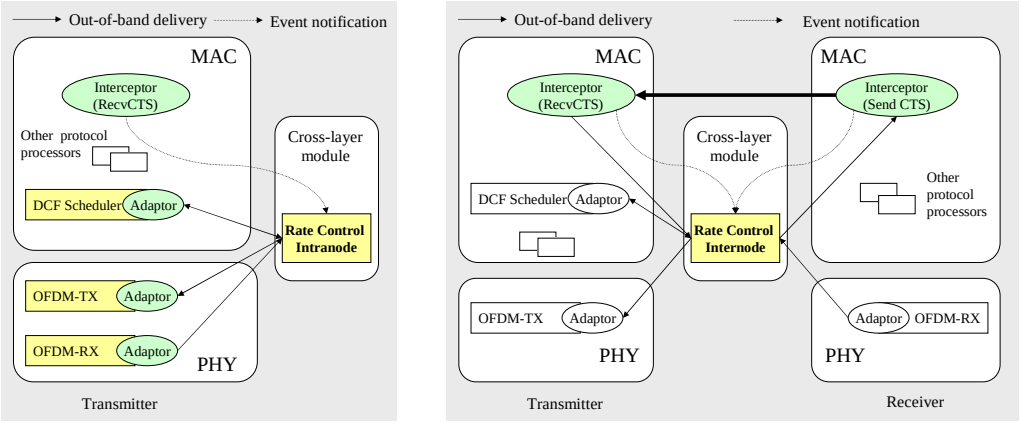
6.2.2 Architecture-based Implementation

The implementations based on our architecture were encouraging in that they reduce to a significant degree changes of the existing protocol processors. Fig. 6.5 shows how we created and modified the protocol processors in the MAC and PHY to implement rate control. We first implemented the Intranode version of rate control. As shown in Fig. 6.5(a), to implement the Intranode version of rate control:

1. A packet processing element was created in Click:

    - to add an Interceptor that notifies the rate control process of the CTS reception.

2. The interfaces of a packet processing element in Click were augmented:



(a) The Intranode version

(b) The Internode version

Figure 6.5: Implementations of rate control based on our architecture

- by attaching an Adaptor that allows rate control to obtain the length of the data packet and to set the selected rate.

3. The interfaces to a set of signal processing blocks in GNU Radio were augmented:

   - by attaching an Adaptor that allows the rate control processor to obtain the channel status, and
   - by attaching an Adaptor that allows the rate control processor to set the selected rate.

4. A rate control processor was created outside the MAC and the PHY:

   - to calculate the rate using the channel status from the PHY, and
   - to set the selected rate to the MAC and the PHY.

Although a set of protocol processors were created and inserted into Click and GNU Radio to implement our architectural components, these components allowed us to

reduce changes in the existing protocol implementations. An Interceptor notifies the Synchronous event handler of the passage of a CTS packet without interactions with the existing packet processing elements. Further the Adaptors augmented the interfaces of the MAC and the PHY simplifying changes to them. The main changes were to expose some variables inside the protocol processors to allow the Adaptor to access the information and simple modification of a protocol processor to allow it to use the rate set by the Adaptor.

We then extended the Intranode version of rate control to the Internode version. As shown in Fig. 6.5(b), implementing the Internode version of rate control required the following steps:

1. A packet processing element was created in Click:
    - to add an Interceptor:
        - · which notifies the rate control process of the CTS transmission, and
        - · which piggybacks the channel status on the CTS packet.
2. An Interceptor in Click was modified:
    - to inform the rate control processor of the channel status piggybacked on the CTS.
3. A rate control processor outside the MAC and the PHY was changed:
    - to piggyback the channel status on the CTS, and
    - to calculate rate using channel status piggybacked on the CTS.

We were able to extend rate control reducing to a significant degree changes to the existing protocol implementations. After implementing and inserting one more Interceptor into Click at the receiver, the Interceptor transparently changes the format of the CTS packet. Then we only needed to modify an Interceptor at the transmitter and the rate control processor outside the MAC and the PHY.

Table 6.1: A comparison of the implementations using metrics

	Conventional implementation	Architecture-based implementation
The number of protocol processors:		
for Intranode version		
- created	1	1 + 1*
- modified	7	3
for Internode version		
- created	0	1
- modified	2	1 + 1*
The number of protocol processors that are involved in delivering:		
for Intranode version		
- Channel status	4	2 + 1*
- Data rate	5	2 + 1*
for Internode version		
- Channel status	5	3 + 1*
- Data rate	5	2 + 1*

*: The number of cross-layer processor: The rate control processor was created and modified outside Click and GNU Radio and also coordinated information exchanges outside Click and GNU Radio

6.2.3 Comparative Analysis

To investigate the benefits and drawbacks of using our architectural framework, we compared the implementation based on our concrete architecture with that using the conventional approach. Table 6.1 presents the metrics we proposed in Section 5.3. We first measured how many protocol processors were created and modified in Click and GNU Radio. This metric measures changes in the existing MAC and PHY implementations and thus allows us to analyze the impact of implementing rate control on the existing MAC and PHY protocols. To show how we implement and extend rate control, we measured the changes required to implement the Intranode version of rate control and then measure the further changes required to extend the implementation for the Internode version. We then measured how many protocol processors were involved to allow rate control to exchange the channel status and the rate. This metric shows how rate control communicates with the protocol proces-

sors in Click and GNU Radio and thus allows us to analyze the interdependencies between protocol processors. To show interdependencies caused by implementation of the Intranode and Internode versions of rate control, we measured the number of protocol processors involved in delivering the information for both the versions of rate control.

In the conventional implementation, rate control obtains the channel status by piggybacking it on the CTS and delivers the rate by piggybacking it on the data packet. Thus when a protocol processor in Click or GNU Radio forwards a packet to another one that runs in a different address space, they need to transform information and piggyback it on a packet. This In-band delivery allows each protocol processor to obtain the channel status and to set the rate at the time of packet delivery without significant processing overhead. However, as shown in Table 6.1, this implementation required a set of modifications in the existing protocol processors and introduced a series of dependencies between them. For example, implementing the Intranode version of rate control required us to modify four protocol processors to allow rate control in the MAC to obtain the channel status from a signal processing block in the PHY. Extending the implementation to the Internode version then required further modifications in the existing protocol processor to allow rate control to obtain the channel status from a receiver. This extension also changed the interdependencies between them. Such In-band delivery required implementations based on the packet handling mechanism provided by each protocol module. Thus it also introduced the changes that were tightly coupled with each protocol module.

In our architecture, however, inserting Interceptors allowed the receiver to deliver the channel status to the transmitter reducing to a significant degree changes to the existing MAC and PHY implementations. Further the Adaptors augmented interfaces of the existing protocol processors by simplifying changes to them. Then the rate control processor communicated with the Interceptors and Adaptors us-

ing the Intranode version of the Out-of-band connector. Thus our architecture allowed the rate control processor to coordinate information exchange between protocol processors outside the protocol module. This approach removed a series of interdependencies that were caused by the In-band delivery of the conventional implementation and limited our concerns mainly to the interactions between the rate control process and the Interceptor and Adaptors. For example, the Intranode version of rate control obtained the channel status by directly communicating with one Interceptor and one Adaptor in the MAC and PHY. Then the Internode version was able to obtain the channel status from a receiver by modification of the existing Interceptor. This shows that our architecture allows rate control to be independent of the infrastructure and to freely change its operation minimizing the impact on existing protocol implementations. Our architecture, however, introduced additional interactions between rate control and the Interceptors to allow rate control to be notified of the time of information delivery and rate selection process.

The evaluation results confirm that our architecture allows the modular implementations of both the existing MAC and PHY protocols and rate control reducing to a significant degree changes in the existing protocol implementations. Thus our architecture provides a useful framework that allows us to implement and extend adaptation without significant impact on existing protocol implementations.

6.3 Refinement for Performance

To evaluate the performance of our architecture, we measure overheads of our architecture. Our architecture can introduce a set of processing overheads to allow communications between rate control and the MAC and PHY protocols over a set of connectors which run in different address spaces. In fact, before the measurements, we expected that most of overheads would come from the processing overhead of the Global and Local connectors that occurs when they route the channel status and the

Table 6.2: Overhead of conventional implementation and our architecture

	Implementation based on architecture	Conventional implementation
MAC and PHY processing time (T_{PROT})	59 ms	
Rate calculation time (T_{CALC})	35 μs	
Communication time (T_{COMM})	548 μs	46 μs
Ratio of adaptation time over Protocol processing time ($R_{ADAPT} = \frac{T_{CALC}+T_{COMM}}{T_{PROT}}$)	0.98%	0.14%
Ratio of communication time over Adaptation time ($R_{COMM} = \frac{T_{COMM}}{T_{CALC}+T_{COMM}}$)	94%	57%

rate to appropriate protocol processors in the MAC and PHY. However, as we will show in detail, our performance measurements showed that most of the overhead comes from interprocess communication that occurs to allow rate control outside a protocol module to communicate with the MAC and PHY. Thus our measurements allowed us to find a way of refining our approach to reduce the communication overhead of our architecture while maintaining to a significant degree the modularity of rate control and the existing protocol implementations.

To understand the key refinement of our approach, we first show our investigation about the performance of our concrete architecture by presenting the communication overhead of the Internode version of rate control. We then show how we refined the implementation of the Internode version of rate control and present the performance of our refined architecture.

6.3.1 Performance of the Concrete Architecture

To identify the mechanisms of our architecture that we need to refine to reduce overhead, we investigated the performance of our architecture by measuring the communication overhead introduced by our architectural components. Table 6.2 shows the communication overhead of our architecture. We measured three differ-

ent processing times for the Intranode version of rate control. We first measured the PHY and MAC processing time (T_{PROT}) that a transmitter and receiver pair spends transmitting a data packet. This protocol processing time also includes the processing time that both the nodes spend exchanging the RTS, CTS, and ACK for a data transmission. We then measured the calculation time (T_{CALC}) that the rate control process at a transmitter spends calculating a rate using the channel status. Finally, we measured the communication time (T_{COMM}) that the rate control process spends exchanging the channel status and the rate. Thus the adaptation time which rate control spends performing adaptation can be calculated by adding the calculation time and the communication time. Since we used the same rate calculation algorithm for both the implementation techniques, a larger communication time increases the adaptation time of each implementation technique.

As further shown in Table 6.2, using these measurements, we calculated the ratio of the adaptation time over the MAC and PHY processing time (R_{ADAPT}). This ratio shows the additional processing time that a transmitter and receiver pair spends selecting an appropriate rate for a data transmission. Thus it presents the overall impact of rate control on the performance of a wireless system. The adaptation time of our architecture is approximately 1% of the protocol processing time of the MAC and PHY. This overhead ratio is larger than that of the conventional implementation. However, such a small processing overhead is insignificant for the performance of a wireless system. Further if we consider that rate control enhances the performance of wireless system by improving the throughput of wireless links, the small overhead will not be a problem.

Table 6.2 further shows the ratio of communication overhead over the adaptation time of rate control (R_{COMM}). This ratio shows how much time rate control spends exchanging the channel status and the rate and thus presents the impact of communication overhead on the processing time rate control requires to perform

Table 6.3: Detail performance measurement of our architecture

Detailed processes in Communication	Processing Ratio
Context Switching	53 %
Message exchange	34 %
Marshall/Unmarshall info.	8 %
Route Info.	5 %

adaptation in detail. In our architecture, rate control spends approximately 94% of adaptation time exchanging channel status and the rate while the conventional implementation spends 57% of its time exchanging information. Thus the communication overhead of our architecture is significant compared to the time which rate control spends calculating the rate.

To identify the mechanisms which led to this overhead, we measured the communication time in detail. Table 6.3 shows that most of the overhead comes from interprocess communication. This interprocess communication was required to allow rate control to communicate with the MAC and PHY which ran in the different address spaces. Interprocess communication introduced context switching between rate control and the MAC and PHY protocols. Further it required exchanging messages after marshalling and unmarshalling the channel status and the rate. These operations made up of approximately 95% of the communication overhead. We expected that a significant overhead would come from the processing overhead of the Global and Local connectors that routes the channel status and the rate to an appropriate protocol processors, but it was only 5% of the overall overhead.

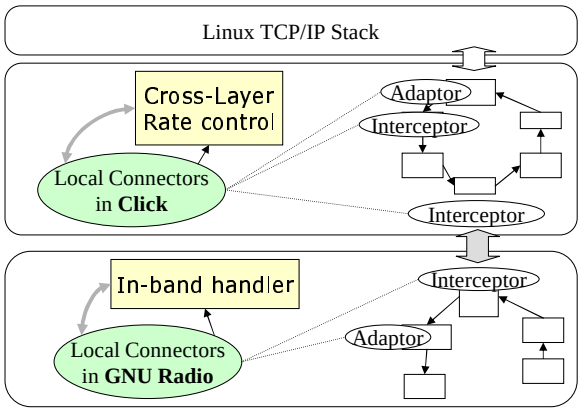


Figure 6.6: Implementation of rate control by using our refined architecture

6.3.2 Refined Concrete Architecture

Our performance measurements allowed us to refine our approach to reduce the communication overhead while supporting a systematic implementation of rate control. The key strategy is to place the rate control processor inside Click and GNU Radio. This refinement allows rate control to run in the same address spaces as Click and GNU Radio and thus eliminate interprocess communication. Fig. 6.6 shows how we refined our concrete architecture in detail. We first implemented the rate control processor as a packet processing element in Click. Then we extended the Local connectors in Click to allow the rate control processor to communicate with Adaptors and Interceptors. Similarly, we implemented a simple cross-layer processor in GNU Radio. This processor piggybacks the channel status and the rate on the existing message format that the MAC and the PHY were using for packet exchange. The processor piggybacks the information by communicating with an Interceptor in GNU Radio and thus allows the MAC to obtain the channel status and to change the rate without interprocess communication.

The refinement required the rate control processor to be modified conforming to the implementation environment provided by Click and GNU Radio. However,

Table 6.4: Communication overhead of refined architecture

		Implementation based on architecture	Conventional implementation	Implementation based on refinement
Communication time	(T_{COMM})	548 μs	46 μs	90 μs
Ratio of adaptation time over Protocol processing time	(R_{ADAPT})	0.98%	0.14%	0.21%
Ratio of communication time over Adaptation time	(R_{COMM})	94%	57%	72%

our new architecture still allows the implementation of rate control to be loosely coupled with the MAC and PHY protocols. The Local connectors were extended to allow the rate control processor to communicate with the Interceptor and Adaptors using the same mechanism that was provided by the Global connectors. Further we were able to use the existing Interceptors and Adaptors without modification. The only significant change was to insert two more Interceptors into Click and GNU Radio to allow the channel status and the rate to be delivered using the In-band delivery mechanism.

Table 6.4 shows that the refined architecture significantly reduced the communication overhead. The communication overhead of our architecture is now slightly larger than that of the conventional implementation. Thus a transmitter and receiver pair is able to execute rate control without additional overhead to perform the adaptation. Further our refined architecture is able to support the implementation without significant performance degradation of rate control itself. We expect that optimizing the implementation of the Local connectors, the Adaptors, and the Interceptors would allow us to further reduce the overhead.

Our measurements confirm that our refined architecture supports the implementation of rate control without significant performance degradation. Further our architecture provides a useful framework also to refine the implementation allowing the modularity of both rate control and the existing protocol implementations.

Chapter 7

Case II: Contention Window Control

Our second case study is contention window control. The purpose of this algorithm is to improve throughput of multihop wireless networks by mitigating unfairness between nodes. A problem of the 802.11 system is that a node can suffer from a significantly lower chance of transmission than that of its neighboring nodes. Such unbalanced transmission reduces the throughput of a link and thus leads to throughput degradation in multihop wireless networks. A set of algorithms have been proposed to improve the performance of multihop wireless networks by mitigating unfairness [63, 64, 65, 66, 67, 68]. Our algorithm is similar to one in [68]. Although detailed operations are different, both the algorithms mitigate unfairness by controlling the contention window of 802.11. As we will show, controlling the contention window allows us to control the probability of transmission at a node. Thus the key idea is that when a node transmits more packets than its neighboring nodes, the MAC adjusts its contention window to allow the neighboring nodes to have better chance of transmissions. Adjusting the contention window balances the throughput of links in a multihop path and eventually improves throughput of

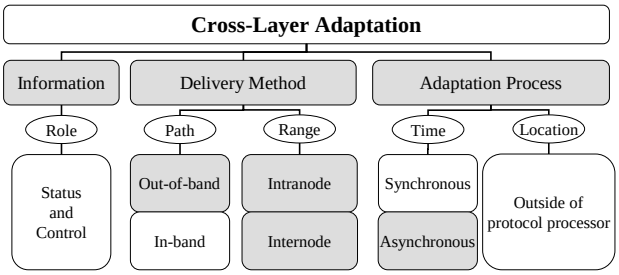


Figure 7.1: Mapping the contention window control to taxonomy

multihop traffic.

We introduce contention window control to evaluate the mechanisms that were not covered by rate control. Contention window control executes an Asynchronous adaptation process that periodically adjusts the contention window, while rate control executed Synchronous process to select an appropriate rate for every packet transmission. Further, contention window control requires interactions between the Network and the MAC layers while rate control required interactions between the MAC and the PHY. Contention window control in the MAC adjusts the contention window by using the number of transmissions monitored by the Network. Contention window control also requires communication between nodes. To detect unfairness in which a node transmits more packets than its neighboring nodes, the MAC exchanges information with its neighboring nodes.

The first goal of this case study is to implement and evaluate the Asynchronous event handler and the Internode version of the Out-of-band connector. Fig. 7.1 shows the taxonomy and the mechanisms we have implemented. We implemented the Intranode version of the Asynchronous event handlers to allow contention window control to periodically update the contention window. Further we implemented the Internode version of the Out-of-band connector and the Asynchronous event handler to allow contention window control to exchange information with neighboring nodes.

The second goal is to evaluate how easily we can implement contention window control using our architectural framework. Thus we also implemented contention window control in a conventional way, in which the MAC directly communicates with the Network layer.

The final goal is to evaluate the performance of our architecture. As in rate control, our performance measurements showed the mechanisms of our architecture that introduce the communication overheads and thus allowed us to refine our approach to support contention window control without significant performance degradation. We implemented contention window control based on the refined architecture and then measured the performance of our architecture.

We begin by discussing the algorithm showing the operation of 802.11, its possible problem and the operation of contention window control. Then we show how we implemented the adaptation both in a conventional way and based on our concrete architecture and present our evaluation by comparing both the implementation techniques. Finally, we show how we refined our architecture to reduce the communication overhead and then present the performance of our architecture.

7.1 Background

We designed and implemented contention window control based on the distributed coordination function (DCF) mode of the 802.11 MAC as implemented within Hydra. We have already presented the basic operation of the DCF protocol in Section 2.1.1. To understand how the DCF works in multihop networks, here we discuss some additional details. Then we show the unfairness problem that leads to throughput degradation in multihop wireless networks. Finally, we show how contention window control mitigates unfairness and thus improves the performance of multihop wireless networks.

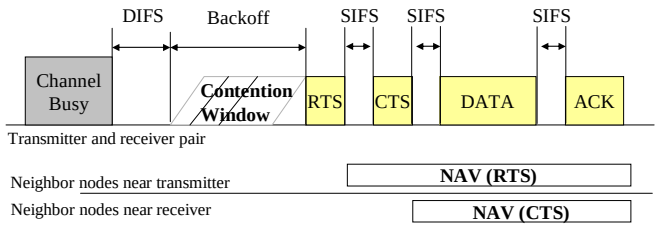


Figure 7.2: DCF mode of 802.11 MAC.

7.1.1 802.11 DCF Protocol

To understand how the 802.11 DCF protocol [31] works in multihop wireless network, we present the operation of the DCF. As shown in Fig. 7.2, before a transmitter sends a packet, it monitors the channel. Then the transmitter defers its transmission until the channel becomes idle to avoid collision with ongoing transmission. Then when the channel has been idle for a distributed inter-frame space (DIFS) interval, the transmitter sends a packet. The DIFS is a fixed interval that is used by all the transmitters. Thus the DIFS allows every transmitter to have the same priority for transmitting packets. However, when multiple nodes have packets to send in their queues, the nodes send their packets at the same time. This leads to collisions between the packets. To reduce the collisions, the DCF de-correlates the transmissions of the nodes by allowing each node to defer an additional random backoff interval. This random time is determined by the "contention window". As further shown in Fig. 7.2, after the DIFS, a node starts the backoff process. A node selects a random backoff time that ranges from zero to the maximum size of the contention window and then starts the backoff timer. When the backoff timer expires, if the wireless channel has remained idle, the transmitter sends a packet. A smaller contention window allows a node to select a smaller backoff time than its neighboring nodes. Thus a node with a smaller contention window transmits packets more aggressively than other neighboring nodes that have a larger contention window.

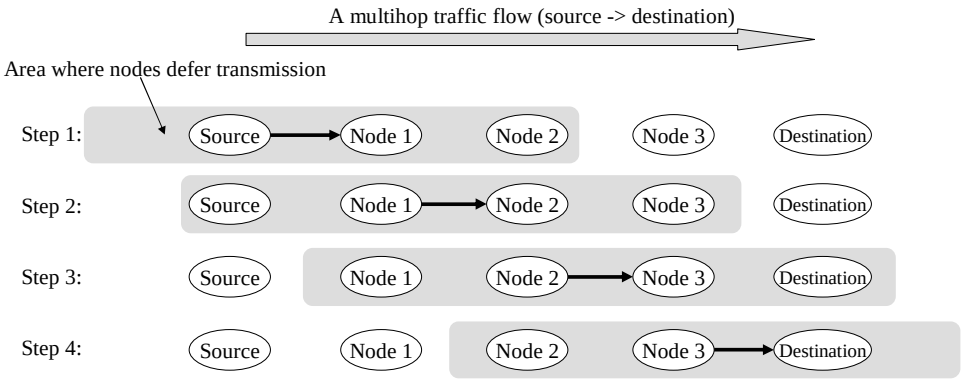


Figure 7.3: A linear multi-hop topology that can lead to unfairness

7.1.2 Unfairness in a Multihop Wireless Network

To understand how contention window control can mitigate unfairness and thus improve the throughput of multihop wireless networks, we first consider how unfairness arises. Fig. 7.3 shows a simple but problematic topology. This linear (chain) topology consists of a source node, a destination node, and three intermediate nodes between them. We assume that the five nodes are evenly spaced at a distance that allows a node to communicate only with its one-hop neighbors. Thus for the source node to transmit packets to the destination node, each packet needs to be delivered to the next-hop node in the chain one at a time. As further shown in Fig. 7.3, in the first step, the source node delivers the packet to node 1. Then node 1 forwards the packet to node 2, its next-hop neighboring node. This process continues until the packet reaches the destination.

In this topology, when a node transmits its packets, some nodes in the multihop path are blocked and defer their transmissions. Ideally, if every node is blocked exactly the same number of times, each node will have the same chance of transmission. This balanced transmission improves the throughput of a multihop traffic flow by allowing each node to deliver packets to its next-hop neighbor node at the same rate. However, the number of blockings is not same for every node in this

topology and thus leads to the unfairness. Detailed analysis is complex; here we give an intuitive explanation. The key problem is that a node in the middle of the multihop path is blocked by the transmissions of nodes on its right-hand side and also on its left-hand side. However the source node is an edge node and thus it is only blocked by those at its right-hand side. We can compare the number of blocking of the source node and node 2. For example, in step 2, the source node is blocked when node 1 (its right-hand side node) transmits packets. Similarly, in step 4, node 2 is blocked when node 3 (its right-hand side node) transmits packets. However, in step 1 and step 2, node 2 is also blocked when the source node and node 1 (its left-hand side nodes) transmit packets.

Due to the unbalanced number of blockings, the source node can transmit packets more aggressively than other nodes, while the intermediate nodes suffer from a lower chance of transmission. Such unbalanced transmission increases the packets remaining in the queues of the intermediate nodes and thus decreases the throughput of a multihop traffic flow. Further the queues of the intermediate nodes can overflow and thus cause packet drops. This packet drop can severely degrade the performance of wireless networks. For example, after a packet drop, TCP retransmits the dropped packet and also cuts down its sending rate assuming that the packet drop is caused by network congestion.

7.1.3 Contention Window Control

The goal of contention window control is to improve throughput of multihop wireless networks by mitigating unfairness. Our key strategy is to slow down the transmissions of a node that transmits more packets than other nodes to balance the transmissions of the nodes. There can be several solutions to address this problem. For example, as shown in [69], TCP can reduce its sending rate and thus balance the transmission of nodes. Further network applications can control their sending rate.

Our contention window control slows down the transmissions of a node by controlling the contention window of the DCF protocol. Our contention window control is similar to one that is proposed in [68] in that both the algorithms balance the transmission by controlling the contention window. Further both algorithms require interactions between the Network and the MAC and also communication between neighboring nodes. However, our main concern was to develop an algorithm that can evaluate our architecture by using an adaptation which executes the Asynchronous process and uses the Intranode and Internode versions of Out-of-band delivery mechanisms, rather than to develop an optimized algorithm. Thus our contention window control might not provide the optimized solution with respect to what criterion is used to detect unfairness and how a node exchanges information with its neighbor nodes. Here we discuss the operation of our contention window control which we have implemented within Hydra.

The key idea of our contention window control is that, when a node transmits more packets than the next-hop node does, the MAC increases its contention window size and slows down its transmission. For example, as illustrated in Fig. 7.3, when the node 1 suffers from a lower chance for transmitting packets than the source node, the source node increases its contention window. Increasing the contention window of the source node allows node 1 to have a better chance for transmitting packets and thus balances the transmission. If the source node still transmits more packets than the next-hop node does, the MAC in the source node keeps increasing its contention window up to a certain maximum size. But when transmissions of both the nodes become balanced, the MAC decreases its contention window. Thus iterating on the adjustment of the contention window allows each node to use an appropriate contention window that can eventually balance transmissions.

We developed our contention window control to evaluate interactions between the Network and the MAC. In our algorithm, contention window control mitigates

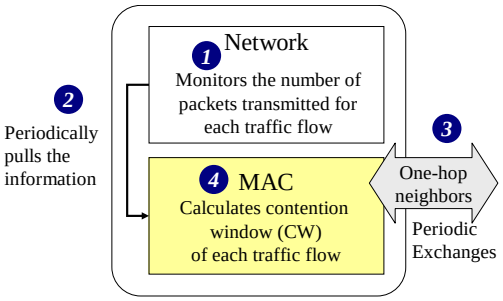


Figure 7.4: Basic operation of contention window control

unfairness of individual multihop traffic flows, each of which has its own source and destination pair. Thus the Network manages the number of transmissions separately based on the source and destination address pair. Then to detect unfairness, the MAC must compare its transmission with that of the next-hop node. Thus the MAC obtains the number of packets transmitted from the Network and also exchanges this information with its one-hop neighboring nodes periodically. Contention window control then executes an Asynchronous adaptation process that periodically updates the contention window, in contrast to rate control which selected an appropriate rate for each packet transmission. Fig. 7.4 shows the operation of contention window control in detail. The steps are:

Step 1: When the Network delivers a packet to the MAC, it reads the source and destination IP addresses of the packet and updates a table that manages the number of transmissions of each multihop traffic flow.

Step 2: The MAC obtains the table from the Network using the Intranode version of the Out-of-band delivery mechanism.

Step 3: The MAC exchanges the table with its neighboring nodes using the Internode version of the Out-of-band delivery mechanism.

Step 4: The MAC periodically updates the contention window using this in-

formation. Then, when it transmits a multihop packet to its next-hop node, it uses the updated contention window to determine its random backoff interval.

7.2 Implementations

To evaluate our architectural framework, we implemented contention window control based on our architecture as well as using a conventional approach. To show feasibility of the implementations, we briefly explain how we implemented contention window control within Hydra. We then show that the implementations are working within Hydra by presenting some experimental results.

7.2.1 Implementation using Hydra

We implemented contention window control to show the feasibility of implementation based on our concrete architecture as well as in a conventional way. We first implemented contention window control in a conventional way. Hydra implements both the Network and the MAC by composing a set of packet processing elements in Click. Thus we created and modified a set of packet processing elements in Click. Then we changed the connection graph of Click to compose the new Network and MAC protocols. We present further detail implementation and evaluation results in Section 7.3.

We then implemented contention window control based on our architectural framework. Fig. 7.5 shows how we implemented contention window control using our concrete architecture. The key is that we created a set of new Global connectors that were not considered by rate control. We first implemented the Intranode version of the Asynchronous event handler as a Global connector, within the Intranode version of the Out-of-band connector that we implemented for rate control. We implemented the Internode version of the Out-of-band connector and the Asynchronous handler as Global connectors. We then extended the Local connectors in Click to allow them

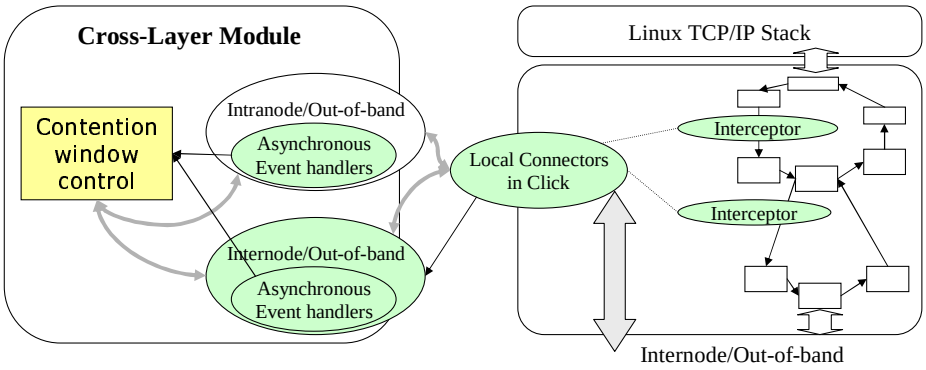
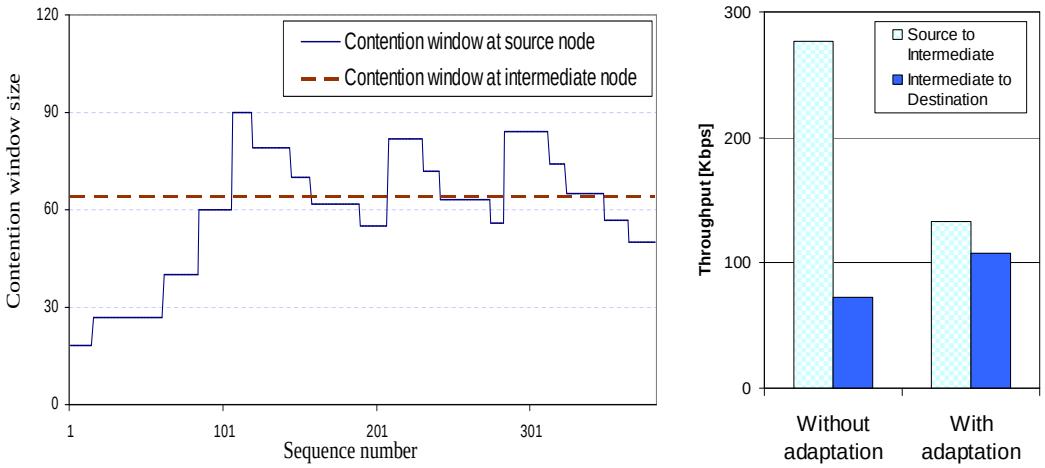


Figure 7.5: Implementation of cross-layer contention window control by using our concrete architecture

to communicate with the new Global connectors. The Local connectors were also extended to send and receive a packet that carries cross-layer information by creating an Internode version of the Out-of-band delivery path. We then implemented a set of Interceptors as packet processing elements in Click and inserted the Interceptors by changing the connection graph. Finally, we implemented the contention window control processor, which communicates with the Network and the MAC using the Global connectors.

7.2.2 Validation of Implementation

To show that our implementations are operational within Hydra, we performed a set of experiments. The goal of the experiments was to show that the implementation of contention window control is working by presenting how contention window control balances the transmissions of nodes in the network, rather than to explore the performance of the algorithm in real world situations. Thus, we used an emulator. This emulator allows Hydra nodes to exchange packets without using the actual PHY and the hardware for the RF front end. Thus it allows us to experiment with a variety of multihop wireless networks reducing the difficulty in controlling the real wireless channels. In this emulated environment, a packet processing element of



(a) Trace of contention window size

(b) Throughput of a multi-hop traffic flow

Figure 7.6: The experimental results of contention window control

Click replaces the actual PHY, providing the MAC with interfaces that were used for communication between the MAC and the PHY. Then the emulated PHY in each Hydra node is connected to a channel emulator. The channel emulator allows us to configure various wireless network topologies and then delivers packets to a set of Hydra nodes based on the configuration.

We then built an experimental setup that can simplify the analysis on how contention window control balances the transmissions of nodes in the network. Our experimental setup consists of three Hydra nodes, each acting as a source node, an intermediate node and a destination node. Using the channel emulator, we created a linear multihop topology in which the source node transmits packets to the destination node over the single intermediate node. This experimental setup consists of two links, one between the source and intermediate nodes and the other between the intermediate and destination nodes. Thus it allows us to show how contention window control adjusts the contention window of the source node to

balance the transmissions between the source and the intermediate nodes. However, in this setup, only the source and intermediate nodes block each other, and thus the unfairness problem does not occur. To introduce unfairness, we purposely set the contention window of the intermediate node larger than that of the source node. Then we created a multihop traffic flow in which the source node continuously transmits packets to the destination.

Fig. 7.6(a) shows how contention window control is working by presenting the changes of the contention window at the source node and the intermediate node. The X-axis is the sequence number of the transmitted packet, and the Y-axis is the size of the contention window. Two lines present the contention window at the source node and the intermediate node. As expected, as the source node transmits packets, it detects unfairness. Thus it increases its contention window until its contention window becomes similar to that of the intermediate node. Then the source node maintains the contention window to balance the transmissions of the source and intermediate nodes. Fig. 7.6(b) shows how contention window control improves throughput of multihop traffic by presenting the throughput of the two individual links in the multihop path. Without contention window control, we observed a severe unfairness between the two links. The intermediate node suffers from a lower chance of transmitting packets, which eventually degrades throughput at the destination node. However, contention window control balanced throughput of both the links and thus improved the throughput of the second link which delivers packets to the destination.

7.3 Evaluation

To evaluate how our architectural framework supports the implementation of contention window control, we compare the implementation based on our concrete architecture with that in the conventional approach. To show our comparison, we

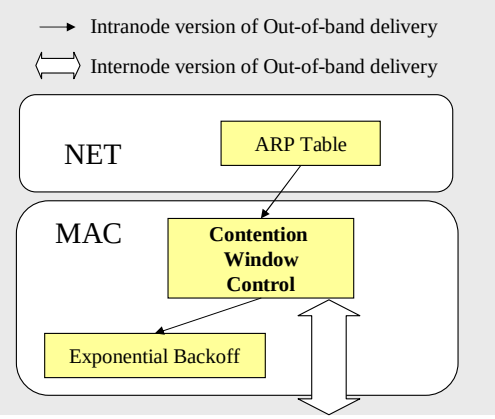


Figure 7.7: A conventional implementation of contention window control

first present detailed implementation and evaluation results for the conventional approach. We then show the detailed implementation and evaluation of our architecture. Finally, we show the comparative analysis on both the implementation techniques.

7.3.1 Conventional Implementation

The conventional implementation required a set of changes to the existing Network and MAC layer implementations. Fig. 7.7 shows how we modified packet processing elements in Click to implement contention window control. Implementing contention window control required the following steps:

1. A packet processing element was created in Click:
   - to allow the MAC to periodically update the contention window:
     - using the number of transmissions as monitored by the Network, and
     - using the number of transmissions delivered from its neighboring nodes.
   - to allow the MAC to periodically exchange the number of transmissions

with its neighboring nodes.

2. Two packet processing elements in Click were modified:

   - to allow the Network to monitor the number of transmissions of each multihop traffic flow,
   - to allow contention window control to obtain the number of transmissions, and
   - to allow the MAC to select a random backoff interval by using the adjusted contention window.

Compared with rate control, contention window control required insignificant changes to the existing Network and MAC implementations. However, the implementation required changes to the existing protocol processors to monitor the number of transmissions and to use the updated contention window. As in rate control, the key problem was that direct communication between the Network and MAC introduced an interdependency between them. Further, to allow contention window control to exchange information with its neighboring nodes, we created an Internode version of the Out-of-band communication path that is dedicated to contention window control.

7.3.2 Architecture-based Implementation

In contrast to the conventional implementation, the implementation based on our architecture reduced to a significant degree changes in the existing protocol implementations. Fig. 7.8 shows how we created and modified packet processing elements in Click to implement contention window control. Implementing contention window control required the following steps:

1. Two packet processing elements were created in Click:

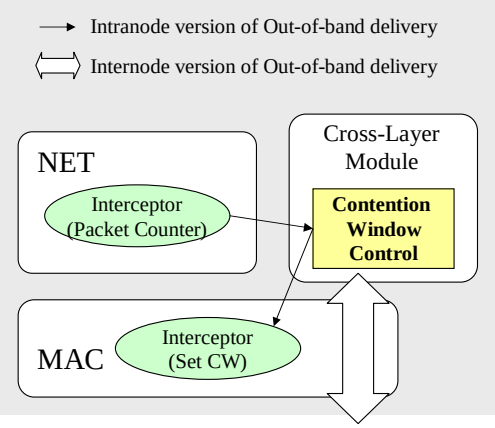


Figure 7.8: Architecture based implementation of contention window control

- to add an Interceptor in the Network:
  - · which monitors the number of transmissions, and
  - · which allows the contention window control processor to obtain the number of transmissions.
- to add an Interceptor in the MAC:
  - · which allows the contention window control processor to set the updated contention window, and
  - · which selects a random backoff interval by using the calculated contention window.

2. A contention window control processor was created outside the Network and the MAC:
   - to periodically update the contention window:
     - · using the number of transmissions monitored by the Network, and
     - · using the number of transmissions delivered from its neighboring nodes.

- to periodically exchange the number of transmission with its neighboring nodes.

Although two Interceptors were created and inserted into Click, our architecture reduced to a significant degree changes in the existing protocol processors. An Interceptor in the Network monitored the packet transmissions and allowed contention window control to periodically acquire this information using the Intranode version of the Out-of-band connector. Then contention window control updated the contention window and delivered the information to another Interceptor in the MAC. This Interceptor transparently applied the updated contention window to every packet transmission[1]. Further, the Internode version of the Out-of-band connector allowed contention window control to exchange information with its neighboring nodes using a similar mechanism to that provided by the Intranode version of the Out-of-band connector.

7.3.3 Comparative Analysis

To investigate the benefits and drawbacks of using our architectural framework, we compared the implementation based on our concrete architecture with that in the conventional approach. Table 7.1 shows the comparison using the metrics we stated in Section 5.3. We first measured how many protocol processors were created and modified in Click. This metric measures changes in the existing Network and MAC implementations and thus allows us to analyze the impact of implementing

[1]The DCF protocol of Click implements a task that selects a random interval using a fixed contention window, as a packet processing element. In this implementation, we were able to allow the DCF to use the contention window calculated by our contention window control by implementing an Interceptor. In our implementation, when a packet is delivered to the MAC, the interceptor sets the contention window for the packet using one calculated by contention window control. Then after the existing element selects a random backoff interval using a fixed contention window, it overrides the selected backoff interval. Another possible implementation was to attach an Adaptor to the existing element and to change the element to use the contention window calculated by contention window control. However, inserting the Interceptor allowed us to further reduce the changes in the existing MAC implementation.

Table 7.1: A comparison of the implementations using metrics

	Conventional implementation	Architecture-based implementation
The number of protocol processors:		
- created	1	2 + 1*
- modified	2	0
The number of protocol processors that are involved in delivering:		
- Contention window	2	1 + 1*
- Number of transmissions of mine	2	1 + 1*
- Number of transmission from neighbors	1	0 + 1*

*: The number of cross-layer processor: The contention window control processor was created outside Click and also coordinated information exchanges outside Click.

contention window control on the existing Network and MAC protocols. We then measured how many protocol processors were involved to allow contention window control to exchange monitored packet transmission and updated contention window. This metric shows how contention window control communicates with packet processing elements in Click and thus allows us to analyze the interdependencies between protocol processors.

Contention window control consists of both the Asynchronous and Synchronous processes. An Asynchronous process periodically updates the contention window while the Synchronous processes monitor packet transmissions and apply the contention window at the time of packet delivery. In the conventional implementation, implementing the Synchronous processes required that the existing protocol processors perform additional processes for contention window control. As shown in Table 7.1, it introduced changes in the existing protocol processors coupled with contention window control. Further, to allow contention window control to exchange information with neighboring nodes, we created a new packet delivery path that is dedicated to contention window control. This communication path introduced an additional change to the MAC, which is tailored to contention window control. How-

ever, in contrast to rate control which exchanged the channel status and the rate using the In-band delivery mechanism, contention window control exchanged the number of transmissions and the contention window using the Out-of-band delivery mechanism that uses a path that is not related to packet delivery. Thus, as further shown in Table 7.1, contention window control even in the conventional implementation communicated directly with the protocol processors without introducing a series of interdependencies between the protocol processors.

Our architecture however allowed us to implement contention window control reducing to a significant degree changes in the existing Network and the MAC. As shown in Table 7.1, we created and inserted two Interceptors in Click. Then we created the contention window control processor outside Click. As in rate control, the window control processor directly communicated with the Interceptors and thus limits our concern mainly to interactions between our architectural frameworks. Further the Internode version of the Out-of-band connector allowed contention window control to exchange the number of transmissions with its neighboring nodes using a similar mechanism to that provided by the Intranode version of the Out-of-band connector. The difference was the destination with which contention window control communicates. The Intranode version of the Out-of-band connector required the name of an Adaptor or an Interceptor while the Internode version required the address of neighboring nodes. However, in our architecture, the contention window control processor and the Interceptors ran in different address spaces. Thus our architecture required additional marshalling and unmarshalling mechanisms to allow contention window control to deliver the updated contention window to the Interceptor within the MAC.

The evaluation results confirm that our architecture allows the modular implementations of both the existing Network and MAC protocols and contention window control reducing to a significant degree changes to the existing protocol

Table 7.2: Overhead of conventional implementation and our architecture

	Implementation base on architecture	Conventional implementation
Adaptation period (T_{PERIOD})	3 s	
Contention window calculation time (T_{CALC})	22 μs	
Communication time (T_{COMM})	500 μs	13 μs
Ratio of Adaptation time over Adaptation period ($R_{ADAPT}=\frac{T_{CALC}+T_{COMM}}{T_{PERIOD}}$)	0.0174%	0.0012%
Ratio of Communication time over Adaptation time ($R_{COMM}=\frac{T_{COMM}}{T_{CALC}+T_{COMM}}$)	96%	37%

implementations. Thus our architecture provides a useful framework that allows contention window control to be implemented without significant impact on the underlying the Network and the MAC protocols.

7.4 Refinement for Performance

Our concrete architecture introduced additional overhead to allow communication between contention window control and the Network and MAC protocols, which ran in a different address space. We refined our architecture to reduce the communication overhead. To understand the key refinement of our approach, we first show an investigation of the performance of our concrete architecture by presenting the communication overhead of contention window control. We then show how we refined the implementation of contention window control and present the performance of our refined architecture.

7.4.1 Performance of the Concrete Architecture

To identify the mechanisms of our architecture which we need to refine to reduce overhead, we investigated the performance of our architecture by measuring the

communication overhead of contention window control. Table 7.2 shows how we measured communication overhead of our architecture. In contrast to rate control which calculated the rate for every packet transmission, contention window control executes an Asynchronous process which periodically updates the contention window. Thus we first measured the period of the Asynchronous process (T_{PERIOD}). The period can vary based on the implementation of contention window control. In our experiment presented in Section 7.2.2, three seconds was an appropriate period that allows contention window control to detect unfairness and to balance the transmissions. Thus we used this time interval in our experiment as the period of the Asynchronous process. We then measured the calculation time (T_{CALC}) which contention window control spends updating the contention window. Finally, we measured the communication time (T_{COMM}) which contention window control spends exchanging the number of transmissions and the updated contention window with the Network and the MAC, and also with its neighboring nodes. To allow contention window control to exchange the number of transmissions with its neighboring nodes, the MAC created a packet that carries the information and then transmitted the packet. As we measured in rate control, the packet processing time which the MAC spends transmitting a packet was significant compared with other communication overhead. To avoid the measurement bias by the packet processing time, we measured only the overhead which the MAC spends creating a packet that carries the number of transmission. The adaptation time which contention window control spends performing overall adaptation can be calculated by adding the calculation time and the communication time. As in rate control, since we used the same algorithm for both the implementation techniques, a larger communication time increases the adaptation time of each implementation technique.

As further shown in Table 7.2, we calculated the ratio of the adaptation time over the period of contention window control (R_{ADAPT}) using these measurements.

Table 7.3: Detail performance measurement of architecture

Detailed processes in Communication	Processing Ratio
Context Switching	47 %
Message exchange	41 %
Marshall/Unmarshall info.	8 %
Route Info.	3 %

This ratio shows the additional processing time which a node spends executing contention window control. Thus it presents the overall impact of contention window control on the performance of a wireless system. The adaptation time of our architecture was approximately 0.02% of the adaptation period. As in the rate control, this overhead ratio is larger than that of the conventional implementation. However, such a small processing overhead is negligible for the performance of a wireless system. Further if we consider that contention window control enhances the performance of multihop wireless networks, the small overhead will not be a problem.

We then calculated the ratio of communication overhead over the adaptation time (R_{COMM}). This ratio shows how much time contention window control spends exchanging the number of transmissions and the updated contention window and thus presents the impact of communication overhead on the processing time which contention window control requires to perform adaptation. As in rate control, the communication overhead of our architecture became significant compared to the calculation time. In our architecture, contention window control spends approximately 96% of its time exchanging information while the conventional implementation spends 37% of its time.

To identify the mechanisms that led to this overhead, we measured communication time in detail. Table 7.3 shows that most of overhead came from interprocess communication, as in rate control. This interprocess communication was

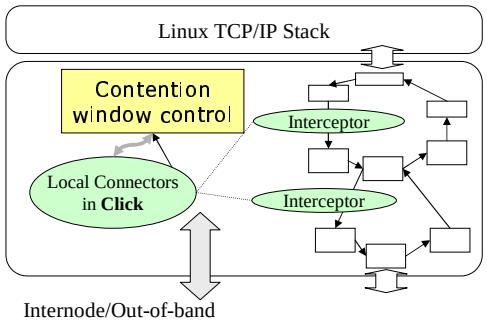


Figure 7.9: Implementation of contention window control by using our refined concrete architecture for performance

required to allow contention window control to communicate with the Network and the MAC which ran in different address spaces. As in the rate control, almost 95% of the communication overhead comes from context switching, message exchange, and marshalling and unmarshalling the number of transmissions and the contention window.

7.4.2 Refined Concrete Architecture

Our performance measurement allowed us to find a way of reducing the communication overhead while supporting the systematic implementation of contention window control. As in rate control, the key strategy is to place the contention window control processor inside Click. This refinement allows contention window control and the Interceptors to run in the same address space and eliminates the interprocess communication. Fig. 7.9 shows how we refined our concrete architecture in detail. We first implemented the contention window control processor as a packet processing element in Click. We then extended the Local connectors to allow contention window control to periodically update the contention window. Finally, as a further performance optimization, we modified the contention window processor and the Interceptors to share the number of transmissions and the updated contention window without mediation of the Local connectors.

Table 7.4: Communication overhead of refined architecture and comparison with conventional implementation

	Implementation based on architecture	Conventional implementation	Implementation based on refinement
Communication time (T_{COMM})	500 μs	13 μs	29 μs
Ratio of Adaptation time over Adaptation period (R_{ADAPT})	0.0174%	0.0012%	0.0017%
Ratio of Communication time over Adaptation time (R_{COMM})	96%	37%	57%

This refinement required the contention window control processor to be modified to conform to the implementation environment provided by Click. However, as in the rate control, the Local connectors were able to allow contention window control to execute its periodic process and to exchange information using the same mechanisms that were provided by the Global connectors. Thus our architecture allowed the implementation of contention window control to be loosely coupled with the existing protocol implementations. We modified the contention window control processor and the Interceptors to share the number of transmissions and the contention window. The modification however did not introduce change to the existing protocol processors and thus still limited our concerns mainly on the interactions between our architectural components.

Table 7.4 shows that the refined architecture significantly reduced the communication overhead. As in the rate control, the communication overhead of our architecture was only slightly larger than that of the conventional implementation. Thus a node was able to execute contention window control without additional overhead to perform adaptation. Further our refined architecture is able to support the implementation without significant performance degradation of contention window control itself. We expect that optimizing the Local connectors would allow further reduction of the overhead.

Our measurements confirm that our refined architecture supports the implementation of contention window control without significant performance degradation. Further our architecture provides a useful framework to refine the implementation allowing the modularity of both contention window control and the existing Network and MAC implementations.

Chapter 8

Case III: A Link-aware Routing Protocol

Our final case study is a link-aware routing protocol. The purpose of this algorithm is to improve the performance of wireless networks by selecting an appropriate multihop path that minimizes transmission time required to deliver packets to a destination. Transmission errors are common in wireless links and bandwidth and latency vary. Thus traditional routing protocols [70, 71, 72] which simply minimize the number of hops between a source and destination pair are inappropriate for wireless networks. A set of link-aware routing protocols have been proposed [73, 74, 75, 76, 51, 77, 78] to address problems of traditional minimum hop-count routing protocols by accounting for the quality of wireless links such as bandwidth, loss rate, and latency. We implemented a link-aware routing protocol [78] that selects multihop paths accounting for two independent link qualities, the data rate used for packet transmission, and the number of retransmissions that occur due to transmission errors. The key idea of the algorithm is to select the wireless links that provide a higher transmission rate and thus speed up packet transmission. At the same time, this routing protocol also considers links which support

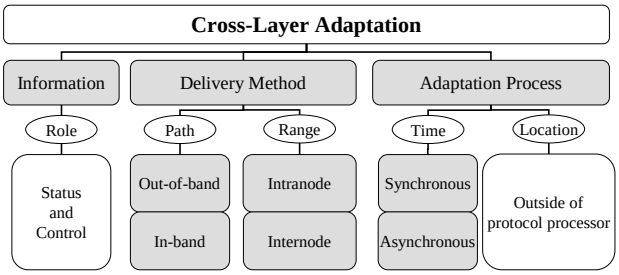


Figure 8.1: Mapping link-aware routing protocol to taxonomy

reliable transmission and thus minimize the number of retransmissions caused by transmission errors. Then it creates a multihop path that minimizes transmission time required to deliver a packet to a destination and thus improves throughput of a multihop wireless link.

The routing protocol requires complex interactions across multiple layers including the Network, MAC and PHY and also introduces interactions across nodes in network. The MAC monitors the rate and the number of retransmissions when it delivers a packet to a neighboring node. Then the link-aware routing protocol in the Network layer periodically obtains this information by communicating with the MAC. In our implementation, rate control in the MAC selects the rate appropriate for each packet transmission and then manages the statistics about the rate. The link-aware routing protocol obtains the rate used for packet transmission by communicating with rate control. Thus this routing protocol introduces interactions with rate control which introduce interactions between the MAC and PHY. Further, to allow the routing protocol to create an appropriate path that minimizes the transmission time to deliver a packet to a destination in the network, the routing protocol in the source node requires the rate and the retransmission counts of all the links in the path. Thus the routing protocol shares information across the nodes in the network. Finally, as an enhancement of operation, our routing protocol periodically transmits probe packets to its neighboring nodes.

The primary goal of this case study is to evaluate how our architecture coordinates complex interactions across multiple layers and across nodes in the network using the mechanism we implemented in our previous case studies. Further we aim to show how our architecture allows interactions between adaptations. Fig. 8.1 shows our taxonomy and the mechanisms required to implement the link-aware routing protocol using our architecture. To allow interactions across multiple layers, we used the Intranode version of the Out-of-band connector. To allow interactions across nodes, we used the Internode version of the In-band and the Out-of-band connectors. We used the Asynchronous event handler to allow the link-aware routing protocol to periodically execute its adaptation process, in addition to the Synchronous event handler that allowed rate control to calculate the rate for every packet transmission. As in our previous case studies, we also implemented the routing protocol using a conventional approach and compared both the implementation techniques to show the benefits and drawbacks of using our architecture. However, the further performance evaluation was not performed in this case study. Since we implemented the link-aware routing protocol using the mechanisms which we implemented in the previous two case studies, the performance of our architecture and the refinement to reduce communication overhead will be fundamentally similar to those in the previous case studies.

We begin by discussing the algorithm in detail by showing traditional minimum hop-count routing protocols, the existing link-aware routing protocols that have been proposed to address the problems of minimum hop-count routing protocols, and the operation of the link-aware routing protocol we implemented. Then we discuss how we implemented the link-aware routing protocol both in a conventional way and based on our concrete architecture. Finally, we present our evaluation by comparing both the implementation techniques.

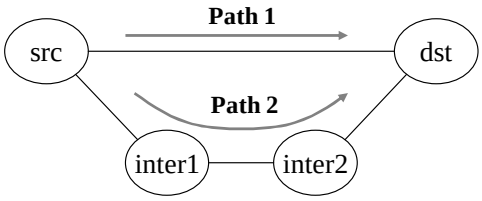


Figure 8.2: A simple topology that can cause throughput degradation of hop-count based routing protocols

8.1 Background

Traditional routing protocols [70, 71, 72] that simply minimize the number of hops required to deliver a packet to a destination are inappropriate for wireless networks. Since bandwidth varies and transmission errors are common in wireless links, to improve performance of multihop wireless networks, a wireless routing protocol can select routes accounting for the quality of links such as bandwidth, loss rate and latency. A set of link-aware routing protocols have been proposed [73, 74, 75, 76, 51, 77] to address problems of traditional minimum hop-count routing protocols. In this case study, we implemented a link-aware routing protocol [78], which selects multihop paths accounting for two independent link metrics, expected transmission time (ETT) [77] and expected transmission count (ETX) [51].

To show how the link-aware routing protocol we implemented addresses the problems of traditional minimum hop-count routing protocols, we first show the problems of minimum hop-count routing protocols. We then present the operations of the ETT and the ETX routing protocols which our routing protocol is based upon. Finally, we show the detailed operation of the link-aware routing protocol we implemented.

8.1.1 Minimum Hop-count Routing Protocols

To show how link-aware routing protocols address the problems associated with traditional routing protocols, we present the basic operation of traditional routing protocols and their possible problems. The metric most commonly used by traditional routing protocols [70, 71, 72] is hop-count. The routing protocols select a route that minimizes the hop-count required to deliver packets to a destination. This approach assumes that links in a network provide reliable transmission and their quality such as bandwidth and latency is likely to be static and further similar to each other. Thus minimizing hop-count minimizes the number of transmissions required to deliver a packet to the destination and thus increases the throughput of the network by reducing the overall transmission time. Minimum hop-count routing protocols were successful to improve throughput of networks made up of reliable and static wired links. However, in wireless networks, transmission errors are common and the quality of wireless links varies significantly. Thus, the minimum hop-count does not always guarantee the minimum transmission time required to deliver packets to a destination.

Fig. 8.2 shows a simple but problematic topology. In this topology, the source node can transmit packets to the destination using two independent multihop paths. The first path is composed of a single link that allows the source to directly deliver a packet to the destination. The other path is composed of three links which allow the source to deliver packets by three intermediate hops. In this topology, a minimum hop-count routing protocol is likely to select the first path which minimizes hop-count to the destination. However the path uses a link that maximizes the distance between the source and destination pair. Thus packets delivered over the link can suffer from frequent packet drops and a low data rate. The packet drops cause retransmissions, and the low rate slows down the packet transmission. Thus the path can significantly increase the transmission time required to deliver a packet

to destination. The second path however is composed of three links that minimize the distance between nodes. Thus the links in the path are likely to provide reliable and high rate transmissions. It allows the source node to deliver a packet to the destination by three high speed transmissions and potentially without retransmission. Thus when the first path suffers from bad channel status, the second path is likely to achieve better throughput than that of the first path.

8.1.2 Existing Link-aware Routing Protocols

A set of link-aware metrics have been proposed [73, 74, 75, 76, 51, 77] to address the problems of minimum hop-count routing protocols. The link-aware routing protocols select multihop paths accounting for the quality of wireless links such as bandwidth, loss rate, and latency. The performance of the routing protocols vary with respect to the link metrics they use. However experimental results [78, 79] show that routing protocols that understand link dynamics are useful to increase throughput of multihop wireless networks. We implemented a link-aware routing protocol using a link metric proposed in [78]. The metric considers two independent link metrics, expected transmission time (ETT) proposed in [77] and expected transmission count (ETX) proposed in [51]. Thus the metric allows a routing protocol to obtain better estimation on the quality of links (we will refer it as enhancement of expected transmission time (EETT)). To show the operation of the EETT routing protocol we implemented, we present the operation of the ETT and the ETX routing protocols which the EETT routing protocol is based upon. Then we show the operation of the EETT routing protocol in detail.

Expected Transmission Time (ETT) Routing Protocol

To address problems of minimum hop-count routing protocols, the ETT routing protocol uses a link metric that estimates the transmission time of links between

pairs of neighboring nodes. Thus it creates multihop paths by selecting the links that minimize the transmission time required to deliver packets to the destination. The ETT routing protocol estimates the transmission time of a link using the rate which a node uses to transmit packets to its neighbor nodes. Thus, as in rate control, it obtains the channel status from the PHY and then calculates the rate appropriate for each link. Fundamentally, a higher rate speeds up transmission and thus reduces the time required to transmit a packet. In the topology shown in Fig. 8.2, assume that the link in the first path suffers from bad channel status and thus a low transmission rate, while the three links in the second path allow high speed transmission. Then the ETT routing protocol can select the second path to reduce the transmission time to deliver a packet to the destination.

Expected Transmission Count (ETX) Routing Protocol

To address problems of minimum hop-count routing protocols, the ETX routing protocol uses a link metric that estimates the number of transmissions (including retransmissions caused by packet drops) required to transmit a packet to its neighboring nodes. Thus it creates multihop paths by selecting the links that minimize the number of transmissions required to deliver packets to the destination. To estimate the number of transmissions, the ETX routing protocol periodically broadcasts a probe packet to its neighboring nodes and then measures the probability that the probe packet fails to reach its neighbor node due to a transmission error. The loss rate of a probe packet allows the ETX routing protocol to estimate the number of retransmissions that can occur at each link. Fundamentally, a higher loss rate increases the number of retransmissions in the MAC and thus increases the time required to transmit a packet to its neighboring node. In the topology shown in Fig. 8.2, assume that the link in the first path suffers from frequent packet drops which lead to more than three retransmissions in the MAC, while the three links

in the second path allow reliable communication which does not introduce any retransmissions. Then the ETX routing protocol selects the second path to deliver the packet to the destination, reducing the number of transmissions and thus the transmission time.

8.1.3 Enhancement of Expected Transmission Time (EETT) Routing Protocol

The goal of the EETT routing protocol [78] is to obtain a better estimate of the transmission time required to deliver a packet to the destination by considering two key qualities of a wireless link, the rate and the retransmission count. Thus the EETT routing protocol estimates transmission time of each link by multiplying the ETT proposed in the ETT routing protocol and the ETX proposed in the ETX routing protocol. However, our strategy for estimating the ETT and the ETX is different from those in the existing the ETT and the ETX routing protocols. Here we discuss our EETT routing protocol as implemented within Hydra.

The key idea of our EETT routing protocol is to acquire the rate and retransmission count by communication mainly with the MAC. To calculate the ETT, the ETT routing protocol calculates the rate appropriate for each link by obtaining the channel status directly from the PHY. Instead, our EETT routing protocol obtains the rate using rate control in the MAC. Rate control selects the rate and manages the statistics on the rate that is used for each link, and the EETT routing protocol periodically obtains the statistics on the rate from the MAC. This approach allows rate control to select an appropriate rate for every packet transmission using up-to-date channel status and thus allows EETT routing protocol to use a better estimate of the rate of each link.

To calculate the ETX, the ETX routing protocol periodically broadcasts a probe packet to its neighboring nodes and then measures the loss rate of its links.

Instead, our EETT routing protocol obtains the retransmission count directly from the MAC. Thus, in our algorithm, the MAC monitors the number of retransmissions required to deliver a packet to its neighbor node and then manages the statistics on the retransmission count of each link. Then the EETT routing protocol periodically obtains the information. This approach allows our EETT routing protocol to obtain the retransmission count without periodic exchanges of the broadcast probe packets. Further this approach allows the EETT routing protocol to obtain a better estimate of the number of transmissions required to deliver a packet to its neighbor node. In the wireless MAC protocol, only "unicast" packets delivered to a particular neighbor node are retransmitted. Thus the MAC monitors the actual retransmission count for each unicast packet transmission. Compared with the ETX routing protocol that estimates the retransmission count of a unicast packet by using broadcast probe packets, our EETT routing protocol obtains the statistics on actual retransmissions that occurred to deliver the unicast packets for which the EETT routing protocol creates multihop paths.

Using the rate and the retransmission count of each link, our EETT routing protocol calculates the EETT and then selects an appropriate path that minimizes the transmission time required to deliver to a destination node in network. Thus the EETT routing protocol requires a node to share its EETT with other nodes in the network. One way of allowing such information sharing is "flooding" [80]. The basic idea is for a node to broadcast the calculated EETT to all of its neighboring nodes. Then each node that receives the EETT forwards it to its neighboring nodes. This forwarding process continues until the EETT reaches all the nodes in the network and thus allow a node to share the calculated EETT with all nodes in the network.

Finally, as an enhancement of our algorithm, the EETT routing protocol also requires interactions between neighboring nodes. In our EETT routing protocol, the MAC monitors the rate and the retransmission count whenever it transmits a unicast

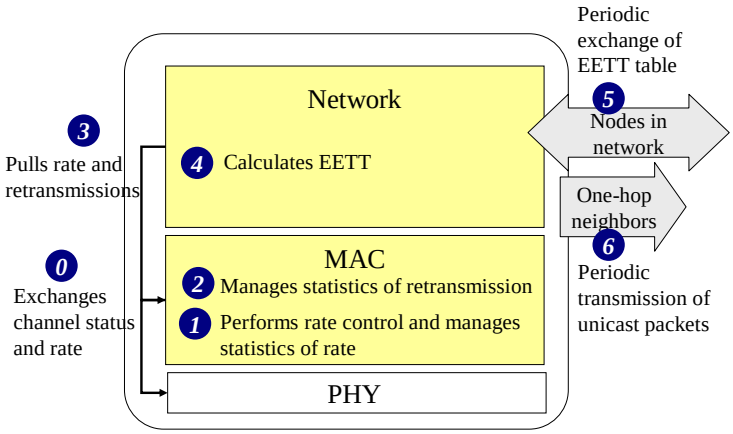


Figure 8.3: Basic operation of contention window control

packet to its neighboring nodes. Thus if a node did not transmit packets for a long time, the MAC might not be able to track changes of the rate and retransmission count of links. This prevents the EETT routing protocol from calculating an up-to-date EETT for some links. To address this problem, our EETT routing protocol periodically transmits unicast packets to its neighboring nodes and allows the MAC to maintain up-to-date rate and retransmission count information.

The EETT routing protocol requires complex interaction across multiple layers and across nodes in network. The EETT routing protocol requires that the MAC performs rate control, which requires interactions between the MAC and PHY. Then the EETT routing protocol in the Network layer periodically obtains the statistics on the rate and the retransmission count by communicating with the MAC. Specifically, the EETT routing protocol obtains the statistics on the rate by communicating with rate control and thus requires interactions also between cross-layer adaptations. Further the EETT routing protocol shares the EETT calculated by a node with all the nodes in the network and transmits unicast probe packets to neighboring nodes. Fig. 8.3 shows the operation of our EETT routing protocol in detail. The steps required to support the EETT routing protocol are:

Step 0: The MAC performs rate control by exchanging the channel status and the rate with the PHY. In our implementation of the EETT routing protocol, we used the Internode version of rate control shown in Section 6. Thus rate control executes its Synchronous process using the Intranode and Internode versions of the In-band delivery mechanism.

Step 1: When the MAC transmits a unicast packet, rate control selects an appropriate rate and then updates the statistics on the rate used for transmitting the packet to the neighboring node.

Step 2: When the MAC transmits and retransmits a unicast packet, it updates the statistics on the retransmission count used for transmitting the packet to a neighboring node.

Step 3: The EETT routing protocol in the Network layer periodically obtains the statistics on the rate from rate control in the MAC and on the retransmission count from the MAC using the Intranode version of the Out-of-band delivery mechanism.

Step 4: The EETT routing protocol periodically executes its Asynchronous process which calculates the EETT of links to its neighboring nodes.

Step 5: The EETT routing protocol periodically broadcasts its EETT table to nodes in a network using the Internode version of the Out-of-band delivery mechanism.

Step 6: The EETT routing protocol periodically transmits unicast packets to its neighboring nodes using the Internode version of the Out-of-band delivery mechanism.

8.2 Implementations

To evaluate how our architecture coordinates the complex interactions across multiple layers, network nodes and adaptations, we implemented the EETT routing protocol both in a conventional approach and based on our concrete architecture. To show feasibility of the implementations, we briefly explain how we implemented the EETT routing protocol within Hydra. We then show that the implementations are working within Hydra by presenting some experimental results.

8.2.1 Implementation using Hydra

We implemented the EETT routing protocol to show the feasibility of implementations based on our concrete architecture as well as in the conventional way. The main goal of the implementation is to evaluate how our architecture coordinates the complex interactions across layers, node and adaptations using the EETT routing protocol that is working within Hydra, rather than to explore the performance of the protocol in real world situations. Thus, as in our second case study, we used an emulated environment. Since the emulator replaces the actual PHY of GNU Radio with the emulated PHY of Click, Click also implements the PHY.

We first implemented the EETT routing protocol in a conventional way. Now all the Network, MAC and PHY are composed of a set of packet processing elements in Click. Thus we created and modified a set of packet processing elements in Click. We then changed the connection graph of Click to compose the new Network, MAC and PHY protocols, which perform the EETT routing protocol using the Internode version of rate control we implemented in our first case study. We present further detail implementation and evaluation results in the next section.

We then implemented the EETT routing protocol based on our architectural framework. Fig 8.4 shows how we implemented the EETT routing protocol using our concrete architecture. The key is that we were able to implement the EETT

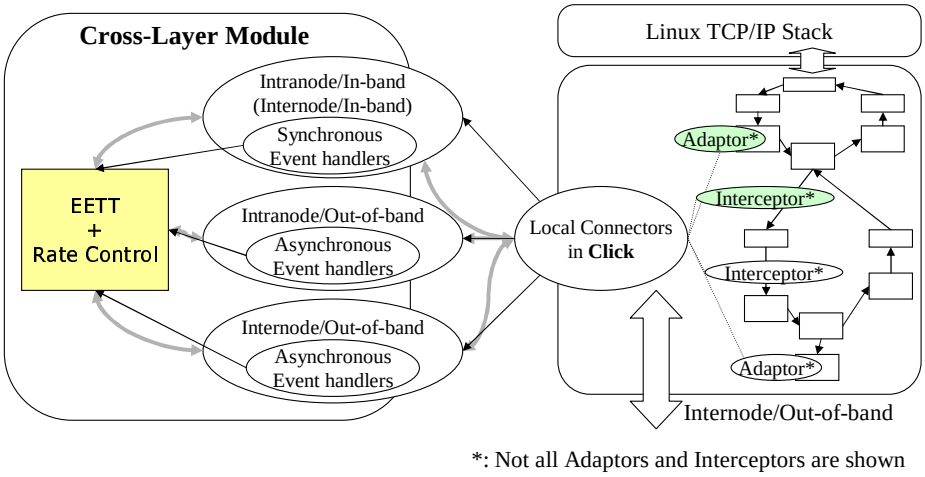
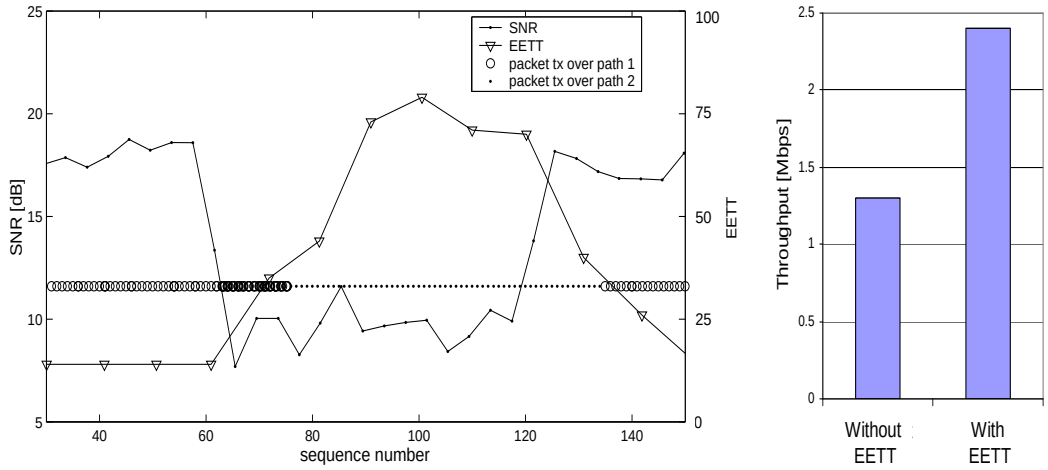


Figure 8.4: Implementation of cross-layer expected transmission time by using our concrete architecture

routing protocol by using the existing Global and Local connectors we implemented in previous case studies. We were also able to reuse the Adaptors and the Interceptors that were implemented for the Internode version of rate control. Thus after we implemented several Adaptors and Interceptors required by the EETT routing protocol as packet process elements in Click, we inserted them into Click by changing the connection graph. We then implemented the EETT processor by extending the existing rate control processor.

8.2.2 Validation of Implementation

To validate that the implementations are operational within Hydra, we performed a set of experiments. As we pointed out, the main goal was to show that the implementation is working within Hydra and thus we used the emulator. Our experimental setup consists of a set of Hydra nodes and the channel emulator. Using the channel emulator, we created a simple multihop topology as presented in Fig. 8.2 and then created a traffic flow in which the source node periodically transmits packets to the destination over the emulated channel. In this experiment, the channel status of



(a) Trace of the channels status and the EETT metric (b) Throughput

Figure 8.5: The experimental results of EETT routing protocol

the link in the first path varied during transmission. Thus, when the channel status of the link is good, the first path supported reliable and high speed transmission. However when the channel status is bad, the path suffered from frequent packet drops even when the source node used the lowest rate supported by the PHY. In contrast, the channel status of the links in the second path remained good during the experiment. They supported reliable transmission which did not cause retransmission and allowed a node to transmit packets using the highest rate supported by the PHY.

Fig. 8.5(a) shows how the EETT routing protocol selects a path to the destination when the channel status of a link in the first path varies with time. The X-axis is the sequence number of the transmitted packet. The left Y-axis shows the SNR, while the right Y-axis shows the calculated EETT. The solid line shows the change of the SNR of the link in the first path and the line with triangles shows the change of the EETT of the link in the first path. Large circles show packets that were transmitted to the destination using the first path while small dots show

packets that were transmitted using the second path of our topology. Finally, the line that is composed of the large circles and small dots shows the summation of the EETT of the three links in the second path and thus the EETT of the second path. As expected, the EETT routing protocol tracks the channel and selects an appropriate path as the channel of the first path varies. As the channel status of the link in the first path became worse, the EETT of the link increased. However, since the channel status of the links in the second path remained the same, when the EETT of the first path became larger than that of the second path, the EETT routing protocol changed the path to the second one. Then when the channel status of the first link improved, the EETT of the link decreased and thus the EETT routing protocol changed the path back to the first one.

Fig. 8.5(b) shows the throughput of the EETT routing protocol and presents a comparison with that of the traditional routing protocol, which does not consider changes of the link qualities. As in the previous experiment, the channel status of the link in the first path changed with time. It stayed good for a time interval and then changed to bad. Then the channel status stayed bad for the same time interval before it changed to good again. In the traditional routing protocol, after it initialized the path to the destination using the first path, it did not change the path during the experiment even though the link of the first path caused frequent packet drops and a lower rate transmission. Thus when the channel status is bad, the routing protocol suffers from low throughput while the EETT routing protocol greatly improved throughput by selecting the second path which provided reliable and high speed transmission.

8.3 Evaluation

To evaluate how our architectural framework supports the implementation of the EETT routing protocol, we compare the implementation based on our concrete

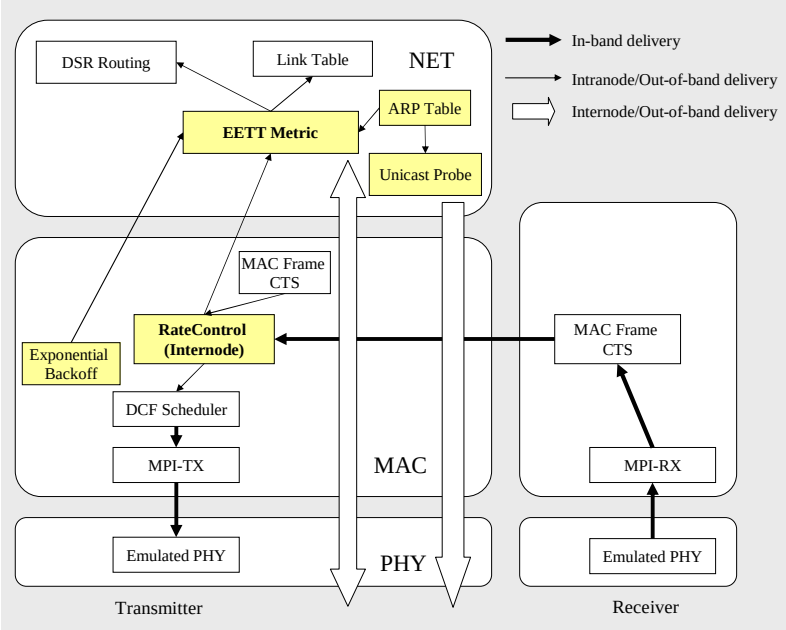


Figure 8.6: A conventional implementation of expected transmission time

architecture with that in a conventional approach. To show our comparison, we first present detailed implementation and evaluation results for the conventional approach. We then show the detailed implementation and evaluation with our architecture. We then present the comparative analysis of both the implementation techniques and show how our architecture coordinates the complex interactions across layers and network nodes and further across cross-layer adaptations.

8.3.1 Conventional Implementation

The conventional implementation required a set of changes to the existing protocol implementations. We implemented the EETT routing protocol using the Internode version of rate control in our first case study. Further we used the existing routing protocol implementation in Click, which selects multihop paths to destinations using the expected transmission count proposed in the ETX routing protocol [51]. Then we

allowed the existing routing protocol to use the calculated EETT instead of the ETX. Thus changes required for the EETT routing protocol were mainly to calculate the EETT of links by using the information in the MAC, to share the calculated EETT across nodes in the network and to transmit unicast probe packets. Fig. 8.6 shows how we created and modified packet processing elements in Click. Implementing the EETT routing protocol required the following steps:

1. Two packet processing elements were created in Click:
    - to allow the EETT routing protocol in the Network layer to periodically calculate the EETT of each link to neighboring nodes:
        · by acquiring the statistics on the retransmission count managed by the MAC,
        · by acquiring the statistics on the rate managed by rate control in the MAC, and
        - by acquiring a table that maps the network IP address to the MAC address.
    - to allow the EETT routing protocol to periodically exchange the EETT with nodes in network.
    - to allow the EETT routing protocol to periodically transmit unicast probe packets to its neighboring nodes.
    - to allow the existing routing protocol to select multihop paths:
        · by using the calculated EETT of links to its neighbor nodes, and
        · by using the calculated EETT from other network nodes.
    - to allow the existing routing protocol to acquire the EETT of a link to its neighbor node.
2. A set of packet processing elements in Click were modified:

- to allow the MAC:
    - to monitor the number of retransmissions that was required to deliver a packet to its neighboring node, and
    - to manage the statistics on the retransmission count of each link.
- to allow the Internode version of rate control in the MAC:
    - to monitor the rate used for each packet transmission, and
    - to manage the statistics on the rate of each link.
- to allow the EETT routing protocol to acquire the address mapping table and then to use this information:
    - to calculate the EETT of links to its neighbor nodes, and
    - to transmit unicast probe packets to its neighbor nodes.

In addition to changes in the MAC and PHY that were required to implement rate control, implementing the EETT routing protocol required a set of changes in the Network and MAC. The key problem was that these additional changes caused additional interdependencies between protocol processors in the Network and MAC. The EETT routing protocol in the Network layer obtained the statistics on the rate and the retransmission count from the MAC. Specifically, the EETT routing protocol obtained the statistics on the rate by communicating with rate control and thus caused rate control to be dependent on the EETT routing protocol. Finally, to allow the EETT routing protocol to exchange the EETT with nodes in network and also to transmit unicast probe packets to its neighboring nodes, the implementation required two independent Internode communication paths that were dedicated to the EETT routing protocol.

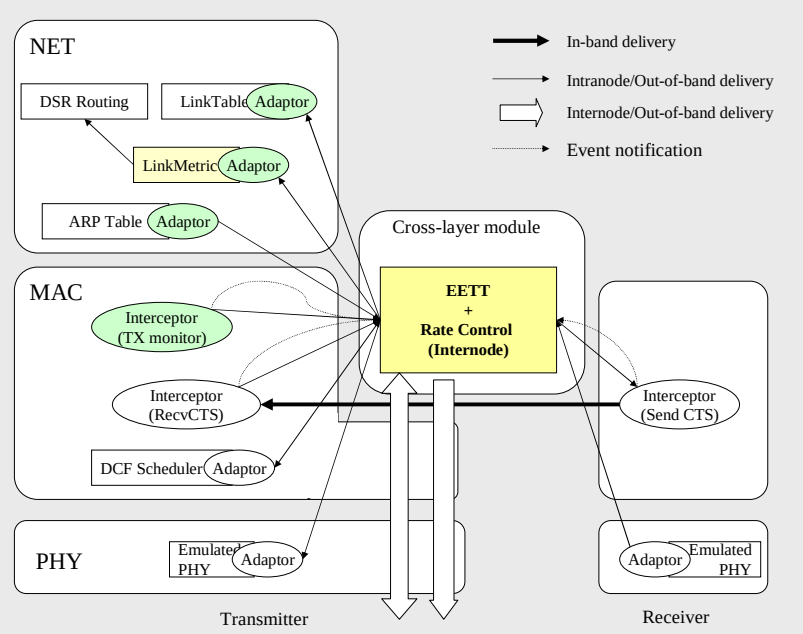


Figure 8.7: Architecture based implementation of expected transmission time

8.3.2 Architecture-based Implementation

Our architecture reduced to a significant degree changes in the existing protocol processors and interdependencies between them. As in the conventional implementation, we implemented the EETT routing protocol using the Internode version of rate control and the existing implementation of the routing protocol in Click. Fig. 8.7 shows how we created and modified packet processing elements in Click. Implementing the EETT routing protocol required the following steps:

1. Two packet processing elements were created in Click:

   - to add an Interceptor in the MAC:

     · which monitors the retransmission count for every packet transmission, and

     · notifies the EETT processor of the retransmission count and the

MAC address of the packet.

- to add a packet processing element in the Network layer:
  - · which allows the existing routing protocol to acquire the EETT of a link to its neighbor node.

2. The interfaces of a set of packet processing elements in Click were augmented:
   - by attaching Adaptors which allow the EETT processor:
     - · to set the calculated EETT of the links to its neighboring nodes to the Network layer,
     - · to set the EETT from nodes in the network to the Network layer, and
     - · to obtain a table that maps the network IP address to the MAC address.
3. The EETT processor was created outside Click by extending the existing rate control processor:
   - to manage the statistics on the rate of each link to its neighboring nodes,
   - to manage the statistics on the retransmission count of each link,
   - to obtain the address mapping table,
   - to periodically calculate the EETT of each link:
     - · by using the statistics on the rate managed by itself, and
     - · by using the statistics on the retransmission count managed by itself,
     - · by using the address mapping table,
   - to periodically exchange the calculated EETT with nodes in the network,
   - to allow the existing routing protocol to select multihop paths:

        - by setting the calculated EETT of each link to its neighboring nodes to the Network layer, and
        - by setting the EETT from nodes in the network to the Network layer,
    - to periodically transmit unicast probe packets by using the address mapping table.

In addition to the Interceptors and Adaptors that were required by rate control, several Interceptor and Adaptors were created and inserted into Click. As in our previous case studies, however, these components did not introduce significant changes in the existing protocol implementations. After an Interceptor in the MAC monitors the retransmission counts, it notifies the EETT processor of the retransmission count for each packet transmission. Then the Adaptors augmented interfaces of protocol processors in the Network layer to allow the existing routing protocol to select multihop paths using the EETT instead of the ETX. An Adaptor was attached to a protocol processor that we created instead of using the existing one. Since the existing routing protocol requested the EETT to its next-hop neighboring nodes using its predefined interface, we created a simple protocol processor based upon this predefined interface and then attached an Adaptor to allow the EETT processor to set the calculated EETT. However this did not introduce change in the existing routing protocol. Then the EETT processor coordinated interactions across multiple layers and executed both rate control and the EETT routing protocol outside the protocol modules. Finally, our architecture allowed the EETT processor to exchange the calculated EETT with nodes in the network and also transmitted unicast probe packets using the mechanism provided by the existing Internode version of the Out-of-band connector.

8.3.3 Comparative Analysis

To investigate the benefits and drawbacks of using our architectural framework, we compared the implementation based on our concrete architecture with that in the conventional approach. Table 8.1 presents the metrics we introduced in Section 5.3. As in our previous case studies, we first measured how many protocol processors were created and modified to measure changes in the existing Network, MAC and PHY implementations. Since the EETT routing protocol was implemented based upon the existing rate control, we measured the number of protocol processors that were created and modified to implement the EETT routing protocol separate from those to implement rate control. We then measured how many protocol processors were involved in delivering information. This metric shows how the EETT routing protocol communicates with a set of protocol processors in the Network, MAC and PHY and thus allows us to analyze the interdependencies between them. We measured the number of protocol processors involved in delivering information required by the EETT routing protocol separate from those by rate control.

In the conventional implementation, the EETT routing protocol required a set of direct communications with protocol processors in the Network and MAC. As shown in Table 8.1, such complex interactions across the Network and MAC required a set of changes in the existing protocol processors. Further, in addition to interdependencies between the MAC and PHY caused by rate control, it introduced interdependencies between the Network and MAC. For example, protocol processors in the MAC were changed to allow to the EETT routing protocol in the Network to obtain the statistics on the rate and the retransmission count. Specifically, the EETT routing protocol acquires the statistics on the rate by communicating with rate control in the MAC and thus introduced changes to the existing rate control coupled with the EETT routing protocol. Finally, the EETT routing protocol shared the calculated EETT with nodes in the network and transmitted unicast probe pack-

Table 8.1: A comparison of the implementations using metrics

	Conventional implementation	Architecture-based implementation
The number of protocol processors:		
for rate control		
- created	1	2 + 1*
- modified	5	2
for EETT routing protocol		
- created	2	2
- modified	3	2 + 1*
The number of protocol processors that are involved in delivering:		
for rate control		
- Channel status	4	3 + 1*
- Data rate	4	2 + 1*
for EETT routing protocol		
- Statistics on data rate	2	0 + 1*
- Statistics on retransmission count	2	0 + 1*
- Retransmission count of a packet	0	1 + 1*
- Address mapping table	3	2 + 1*
- EETT table of mine	3	3 + 1*
- EETT table from neighboring node	2	1 + 1*
- Unicast probe	1	0 + 1*

*: The number of cross-layer processor: The EETT processor were implemented outside Click, and also coordinated information exchanges outside Click.

ets to its neighbor nodes by using two independent communication paths dedicated to the EETT routing protocol. Such internode communication paths introduced additional changes in the Network layer tailored to the EETT routing protocol.

Our architecture however reduced to a significant degree changes in the existing protocol processors. After inserting the Interceptors and Adaptors required by rate control in the MAC and PHY, we created and inserted several more Interceptors and Adaptors into the MAC and Network layers. Then the EETT processor communicates with the Interceptors and Adaptors across multiple layers using the Global connectors. Thus our architecture allowed the EETT routing protocol to coordinate the information exchanges across the Network, MAC and PHY without significant changes in the existing protocol processors and thus limited the interdependencies between them. Further, we implemented the EETT processor which performs both rate control and the EETT routing protocol outside Click. This implementation allowed us to further reduce changes in the existing protocol processors and thus the interdependencies between them. We were able to extend the rate control processor to manage the statistics on the rate. Thus the EETT processor was able to obtain the information without interactions with the protocol processor in the MAC. Further after the EETT processor acquired the address mapping table, it was able to use this information to transmit unicast probe packets to its neighboring nodes. Thus transmitting unicast probe packets did not require further interactions with the existing protocol processors. Finally, our architecture allowed the EETT processor to share the calculated EETT with nodes in the network and to transmit unicast probe packets using the mechanism provided by the existing Internode version of the Out-of-band connector.

The evaluation results confirm that our architecture coordinates complex interactions across multiple layers and across network nodes by allowing modular implementation of both the existing protocol layers and the EETT routing protocol.

Further our architecture reduces changes in the existing protocol implementations and interdependencies between them by coordinating rate control and the EETT protocol outside of the protocol module. Thus our architecture provides a useful framework that allows the EETT routing protocol to be implemented without significant impact on the underlying protocol implementations.

The performance evaluation was not performed for this case study. Since we implemented the EETT routing protocol using the mechanisms which we implemented in the previous two case studies, the performance of our architecture and the refinement to reduce communication overhead would be similar to those in the previous case studies. For example, to reduce the communication overhead caused by interprocess communication between the EETT processor and Click, we can implement the EETT processor to run in the address space where Click runs. Then the existing Local connectors in Click would allow the EETT processor to interact across layers and across nodes in networks while maintaining the modularity of the EETT processor.

Chapter 9

Contributions, Future Work and Conclusions

Our main contribution is to demonstrate that a new software architecture is able to support the implementation of a wide variety of cross-layer adaptations while maintaining the advantages of modularity found in current layered network architectures. Further we showed that such a systematic framework is able to reduce the complexity of implementation for cross-layer adaptations without significant impact on the existing protocol implementations. Detailed contributions are as follows:

- **Taxonomy:** We developed a taxonomy that describes the design space of cross-layer adaptations. Contributions of our taxonomy are:
  - Generalization and classification of the operations of a wide variety of cross-layer adaptations.
  - Creation of a common language that can be used to analyze our design space.
  - Creation of a framework that can be used to develop a cross-layer architecture.

- **Conceptual architecture:** We developed a conceptual architecture that supports systematic implementation of cross-layer adaptations. Contributions of our conceptual architecture are:
  - Support of the key mechanisms that are required to implement the cross-layer adaptations described by our taxonomy.
  - Creation of a systematic framework that allows the modular implementations of existing protocols and cross-layer adaptations.
  - Creation of a generic architecture that can be used to derive a wide set of cross-layer architectures.
- **Concrete architecture:** We extended our conceptual architecture and developed a concrete architecture to implement our architectural framework on real wireless systems. Contributions of our concrete architecture are:
  - Support of the implementation of our architectural framework within Hydra by considering the detailed implementation issues.
  - Creation of a refined architecture that can reduce the communication overhead of our architecture.
- **Evaluation:** We implemented and evaluated our architecture by performing three case studies. Contributions of our case studies are:
  - Implementation of three cross-layer adaptations and the key mechanisms provided by our architecture.
  - Validation of our architecture by showing that it supports the implementation of a wide set of cross-layer adaptations while maintaining to a significant degree the modularity of the existing protocol implementations and cross-layer adaptations.

- Measurement of the performance of our architecture, which allows us to refine our concrete architecture to reduce the communication overhead.
- Validation of our refined architecture by showing that it supports the modular implementation of cross-layer adaptations without significant performance degradation.

9.1 Future Work

We can make several proposals as future work to extend our study. There are two main directions for this work. One direction is to extend our taxonomy and architecture to provide further mechanisms that describe and support the implementation of cross-layer adaptations. The second direction is to evaluate the detailed mechanisms of our architecture by performing more case studies. Here we discuss the work we could perform in each direction.

Extension of the taxonomy and architecture: The first direction is to extend our taxonomy and architecture to provide further key mechanisms that describe and support the implementation of cross-layer adaptations. For example, we could extend our framework to provide additional Internode delivery mechanisms.

In computer networks, there are three key delivery mechanisms that allow a node to exchange information with other nodes in network. The first mechanism is unicasting. This mechanism allows a node to transmit packets that will be received by a specific node. The second one is broadcasting. It allows a node to transmit packets that will be received by every node in the network. In practice, the scope of broadcast is limited to neighboring nodes that a node can directly reach, but the propagation of packets can be expanded by allowing a node to forward a received broadcast packet to its neighboring nodes. The final mechanism is multicasting. In contrast to unicasting and broadcasting, it allows a node to transmit packets that

will be received by a set of nodes in the network. As in unicasting, it allows a node to transmit packets to specific destinations. Further, as in broadcasting, it allows a node to transmit packets that will be received by multiple nodes.

In our taxonomy and architecture, path control describes an Internode delivery mechanism that allows a node to deliver information to a specific destination by sending a unicast packet. Area control describes another mechanism that allows a node to propagate information to its neighboring nodes by sending a broadcast packet. However, some cross-layer adaptations may require a node to deliver information to a set of nodes in network by using multicast packets. In practice, multicasting provides a mechanism in which a set of receiver nodes join in a "multicast group". Then it allows a node to transmit packets that will be received by the nodes in the multicast group. Thus we could extend our taxonomy and architecture to support such a "group control" mechanism in addition to the existing path and area control mechanisms.

Evaluation of detailed mechanisms: The second direction is to further evaluate the detailed mechanisms of our architecture by performing more case studies. First, we could evaluate the Internode delivery mechanisms that were not covered by our three case studies.

We could perform a case study that evaluates the multihop version of the path control mechanism. This case study would allow us to evaluate how the Internode version of the In-band connector delivers information to a specific node that can be reached by several intermediate hops. Further we could evaluate how the Internode version of the In-band delivery mechanism can be used for area control. For example, in the destination sequenced distance vector (DSDV) routing protocol [70], which is one of the proactive wireless routing protocols, to maintain the consistency of routing tables in a dynamically varying topology, each node periodically broadcasts its routing table to its neighboring nodes. Thus, we could extend our contention

window control to exchange the number of transmissions with its neighboring nodes by piggybacking this information on the periodic updates of the DSDV. This case study would allow us to evaluate how our architecture supports the area control mechanism reducing overheads that occurred to create and transmit a packet that carries cross-layer information.

Second, it would also be interesting to implement our architectural framework on another wireless system than Hydra. For example, we could implement a cross-layer adaptation that communicates with the MAC and PHY which are running on a network interface card. The network interface card usually implements these protocols in a firmware that is embedded in a small hardware device. Further it implements some time-critical tasks as hardware logic. Thus this case study would show how we can refine our concrete architecture, specifically, the Interceptors, the Adaptors and the Local connectors which are implemented conforming to the new implementation environment.

9.2 Conclusions

Cross-layer adaptation is a useful protocol design technique that optimizes the performance of wireless networks by using information from many layers of the network. The key problem of cross-layer adaptation however is that the implementation of cross-layer adaptations introduces complex interactions between layers. These ad-hoc implementations not only compromise the modularity of the layered architecture but also introduce substantial changes in existing protocol implementations that are tailored to particular cross-layer adaptations.

To address these problems, we proposed a new software architecture that provides a systematic framework for the implementation of cross-layer adaptations. We first created a taxonomy that describes the design space of cross-layer adaptations. Then we developed a conceptual architecture that supports the implementation of

a wide variety of cross-layer adaptations while preserving the modularity of the existing protocols. We extended the conceptual architecture and developed a concrete architecture to implement our architectural framework within Hydra. Finally, we evaluated the proposed architecture by performing three case studies. The case studies showed that the proposed software architecture is able to support the implementation of a wide set of cross-layer adaptations while reducing changes in the existing protocol implementations. Further our performance measurements allowed us to refine our approach to reduce the communication overhead of our architecture. We showed that this refined architecture can reduce the communication overhead while maintaining to a significant degree the modularity of cross-layer adaptations and the existing protocol implementations.

One important lesson we learned from this study is to gain insight into what concrete architecture is most desirable to balance the trade-off between maintaining the modularity and achieving performance efficiencies. Our study showed the three implementation techniques and presented pros and cons of each technique. First, the conventional implementations can minimize the processing overhead of cross-layer adaptation, while they lead to a set of changes that introduce interdependencies between protocol layers. Second, our concrete architecture maintains the modularity of the existing protocol implementations as well as cross-layer adaptations. Such a loosely coupled architecture is also useful to coordinate complex interactions across layers and to support multiple adaptations in a system. However, it can introduce several overheads to allow the communication between adaptation and the protocol modules that are running in their own address spaces. In contrast, our refined concrete architecture stands in the middle of these two extreme implementation techniques. Our refined architecture is able to reduce to a significant degree the overheads of our architecture, but it is still able to support the modular implementation of adaptation, reducing changes in the existing protocol implementations.

Thus we expect that our refined concrete architecture can be a practical solution that supports a systematic implementation of cross-layer adaptations balancing the trade-off between modularity and performance efficiency of wireless systems.

Bibliography

[1] Computer Science and Telecommunications Board, National Research Council, *Realizing the Information Future: The Internet and Beyond*. Washington, D.C.: National Academy Press, 1994.

[2] Z. J. Haas, "Design Methodologies for Adaptive and Multimedia Networks," *IEEE Communications Magazine*, vol. 39, pp. 106–107, Nov. 2001.

[3] V. Raisinghani and S. Iyer, "Cross-Layer Design Optimizations in Wireless Protocol Stacks," *Computer Communications*, vol. 27, pp. 720–725, May 2004.

[4] M. Conti, G. Maselli, G. Turi, and S. Giordano, "Cross-Layering in Mobile Ad Hoc Network Design," *Transactions on IEEE Computer*, vol. 37, pp. 48–51, Feb. 2004.

[5] C. E. Jones, K. M. Sivalingam, P. Agrawal, and J.-C. Chen, "A Survey of Energy Efficient Network Protocols for Wireless Networks," *Wireless Networks*, vol. 7, pp. 343–358, Apr. 2001.

[6] H. Zimmerman, "The OSI Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, Apr. 1980.

[7] K. Pentikousis, "TCP in Wired-cum-Wireless Environments," *IEEE Communications Surveys*, pp. 2–14, Fourth Quarter 2000.

[8] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz, "Improving TCP/IP Performance over Wireless Networks," in *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, Berkeley, CA, Nov. 1995, pp. 2–11.

[9] L. L. Peterson and B. S. Davie, *Computer Networks: A System Approach*. San Fransisco, CA: Morgan Kaufmann Publishers, 2000.

[10] V. Kawadia and P. R. Kumar, "A Cautionary Perspective on Cross Layer Design," *IEEE Wireless Communications*, vol. 12, pp. 3–11, Feb. 2005.

[11] V. Srivastava and M. Motani, "Cross-Layer Design: A Survey and the Road Ahead," *IEEE Communications Manazine*, vol. 43, pp. 112–119, Dec. 2005.

[12] G. Wu, Y. Bai, J. Lai, and A. Orielski, "Interactions between TCP and RLP in Wireless Internet," in *Proceedings of IEEE Global Telecommunications Conference (Globecom)*, Rio de Janeiro, Brazil, Dec. 1999, pp. 661–666.

[13] M. Conti, S. Giordano, G. Maselli, and G. Turi, "MobileMAN: Mobile Metropolitan Ad Hoc Networks," *Lecture Notes in Computer Science*, vol. 2775, pp. 169–174, 2003.

[14] V. Raisinghani and S. Iyer, "Cross-Layer Feedback Architecture for Mobile Device Protocol Stacks," *IEEE Communications Manazine*, vol. 44, pp. 85–92, Jan. 2006.

[15] R. Winter, J. H. Schiler, N. Nikaein, and C. Bonnet, "CrossTalk: Cross-Layer Decision Support Based on Global Knowledge," *IEEE Communications Manazine*, vol. 44, pp. 93–99, Jan. 2006.

[16] G. Carneiro, J. Ruela, and M. Ricardo, "Cross-Layer Design in 4G Wireless Terminals," *IEEE Wireless Communications*, vol. 11, pp. 7–13, Apr. 2004.

[17] W. Yuan, K. Nahrstedt, S. V. Adve, D. L. Jones, and R. H. Kravets, "Design and Evaluation of a Cross-Layer Adaptation Framework for Mobile Multimedia Systems," in *Proceedings of SPIE/ACM Multimedia Computing and Networking (MMCN'03)*, Santa Clara, CA,, Jan. 2003.

[18] R. H. Ivan Bowman and N. Brewster, "Linux as a Case Study: Its Extracted Software Architecture," in *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, May 1999, pp. 555–563.

[19] D. Perry, "Generic Architecture Descriptions for Product Lines," in *Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families (ARES II)*, Las Palmas de Gran Canaria, Spain, Feb. 1998, pp. 51 – 56.

[20] K. Mandke, S.-H. Choi, G. Kim, R. Grant, R. Daniels, W. Kim, R. Heath, and S. Nettles, "Early Results on Hydra: A Rapid SDR Prototyping Platform," in *Proceedings of IEEE vehicular Technology Conference (VTC)*, Dublin, Ireland, Apr. 2007.

[21] I. Haratcherev, J. Taal, K. Langendoen, R. Lagendijk, and H. Sips, "Adaptive Link Layer Strategies for Asymmetric High-Speed Wireless Communications," *IEEE Transactions on Wireless Communications*, vol. 1, pp. 429–438, July 2002.

[22] ——, "Automatic IEEE 802.11 rate control for streaming applications," *Wireless Communications and Mobile Computing*, vol. 5, pp. 421–437, June 2005.

[23] D. Qiao, S. Choi, and K. G. Shin, "Goodput Analysis and Link Adaptation for IEEE 802.11a Wireless LANs," *IEEE Transactions on Mobile Computing*, vol. 1, pp. 278–292, Oct. 2002.

[24] J. del Prado Pavon and S. Choi, "Link Adaptation Strategy for IEEE 802.11 WLAN via Received Signal Strength Measurement," in *Proceedings of IEEE International Conference on Communications (ICC'03)*, Anchorage, AK, May 2003, pp. 1108–1113.

[25] B. Chen, F. Fitzek, J. Gross, R. Grunheid, H. Rohling, and A. Wolisz, "Framework for Combined Optimization of DLC and Physical Layer in Mobile OFDM Systems," in *Proceedings of the 5th International OFDM-Workshop*, Hamburg, Germany, Sept. 2001.

[26] I. Haratcherev, K. Langendoen, R. Lagendijk, and H. Slips, "Hybrid Rate Control for IEEE 802.11," in *ACM International Workshop on Mobility Management and Wireless Access Protocols (MobiWac'04)*, Philadelphia, PA, Oct. 2004.

[27] M. Lampe, H. Rohling, and W. Zirwas, "Misunderstandings About Link Adaptation For Frequency Selective Fading Channels," in *IEEE International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC'02)*, Lisboa, Portugal, Sept. 2002, pp. 710–714.

[28] B. Sadeghi, V. Kanodia, A. Sabharwal, and E. Knightly, "Opportunistic Media Access for Multirate Ad Hoc Networks," in *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, Atlanta, GA, Sept. 2002.

[29] M. Lacage, M. H. Manshaei, and T. Turletti, "IEEE 802.11 Rate Adaptation: A Practical Approach," in *Proceedings of the 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM'04)*, Venice, Italy, Oct. 2004, pp. 126–134.

[30] N. V. Gavin Holland and P. Bahl, "A Rate-Adaptive MAC Protocol for Multi-

Hop Wireless Networks," in *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, Rome, Italy, July 2001.

[31] *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification*, IEEE Std. 802.11, 1997.

[32] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris, "Link-level Measurements from an 802.11b Mesh Network," in *Proceeding of the Special Interest Group on Data Communication (SIGCOMM)*, New York, NY, Aug. 2004, pp. 121–132.

[33] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," in *Symposium on Operating Systems Principles*, Kiawah Island Resort, SC, Dec. 1999, pp. 217–231.

[34] J. Ueyama, S. Schmid, G. Coulson, G. Blair, A. T. Gomes, A. Joolia, and K. Lee, "A Re-configurable Component Model for Programmable Nodes," in *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops (ICDCSW'04)*, Tokyo, Japan, Mar. 2004, pp. 375 – 380.

[35] T. Scholer and C. Muller-Schloer, "Design, Implementation and Validation of a generic and reconfigurable Protocol Stack Framework for mobile Terminals," in *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops (ICDCSW'04)*, Tokyo, Japan, Mar. 2004, pp. 362–367.

[36] S.-H. Choi, D. Perry, and S. Nettles, "A Software Architecture for Cross-Layer Wireless Network Adaptations," in *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, Vancouver, BC, Canada, Feb. 2008, pp. 281–284.

[37] D. Musser, "Generic Programming," May 2003, http://www.cs.rpi.edu/ musser/gp/.

[38] D. Perry and A. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, pp. 40–52, Oct. 1992.

[39] N. Mehta, N. Medvidovic, and S. Phadke, "Towards a Taxonomy of Software Connectors," in *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, Limerick, Ireland, June 2000, pp. 178–187.

[40] T. Stevens, B. Davies, and A. Fapojuwo, "Cross Layer Signaling in Wireless Ad-hoc Networks Using Embedded Computers," in *Wireless 2005*, Calgary, Alberta, Canada, July 2005, pp. 62–67.

[41] K. Chen, S. H. Shah, and K. Nahrstedt, "Cross-Layer Design for Data Accessibility in Mobile Ad hoc Networks," *Wireless Personal Communications*, vol. 21, pp. 49–76, Apr. 2002.

[42] A. Kopke, V. Handziski, J.-H. Hauer, and H. Karl, "Structuring the Information Flow in Component-Based Protocol Implementations for Wireless Sensor Nodes," in *Proceedings of Work-In-Progress Session of the 1st European Workshop on Wireless Sensor Networks (EWSN)*, Berlin, Germany, Jan. 2004, pp. 41–45.

[43] H. Aiache, V. Conan, G. Guibe, J. Leguay, C. L. Martret, J. Barcelo, L. Cerda, J. Garcia, R. Knopp, N. Nikaein, X. Gonzalez, A. Zeini, O. Apilo, A. Boukalov, J. Karvo, H. Koskinen, L. Bergonzi, J. Diaz, J. Meessen, C. Blondia, P. D. Cleyn, E. V. de Velde, and M. Voorhaen, "WIDENS: Advanced Wireless Ad-Hoc Networks for Public Safety," in *Proceedings of the IST Mobile and Wireless Communications Summit 2005*, Dresden, Germany, 2005.

[44] P. J. Marrón, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel, "TinyCubus: A Flexible and Adaptive Framework for Sensor Networks," in *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, Istanbul, Turkey, Jan. 2005, pp. 278–289.

[45] A. Lachenmann, P. J. Marrón, D. Minder, and K. Rothermel, "An Analysis of Cross-Layer Interactions in Sensor Network Applications," in *Proceedings of the Second International Conference on Intelligent Sensors, Sensor Networks & Information Processing (ISSNIP 2005)*, Melbourne, Australia, Dec. 2005, pp. 121–126.

[46] E. Blossom, "Universal Software Radio Peripheral," Mar. 2006, http://comsec.com/wiki?UniversalSoftwareRadioPeripheral.

[47] ——, "GNU Radio," Aug. 2006, http://www.gnu.org/software/gnuradio/index.html.

[48] *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: High-speed Physical Layer in the 5 GHz Band*, IEEE 802.11 WG, Part 11 Std. 802.11, Sept. 1999.

[49] R. V. Nee and R. Prasad, *OFDM for Wireless Multimedia Communications*. London, UK: Artech House Publishers, 2000.

[50] B. Braem, "Implementation and Evaluation of Ad-hoc Distance Vector Routing," Master's thesis, Universiteit Antwerpen, June 2005.

[51] D. D. Couto, D. Aguayo, J. Bicket, and R.Morris, "A High-Throughput Path Metric for Multi-Hop Wireless Routing," in *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, San Diego, CA, Sept. 2003.

[52] W. R. Stevens and G. R. Wright, *TCP/IP Illustrated, Volume 2: The Implementation*. Reading, MA: Addison-Wesley, 1995.

[53] R. Braden, T. Faber, and M. Handley, "From Protocol Stack to Protocol Heap - Role-Based Architecture," *ACM SIGCOMM Computer Communication Review*, vol. 33, pp. 17–22, Jan. 2003.

[54] L. Berlemann, R. Pabst, M. Schinnenburg, and B. Walke, "Reconfigurable Multi-mode Protocol Reference Model Facilitating Modes Convergence," in *Proceedings of the 11th European Wireless conference 2005 (EW 2005)*, Nicosia, Cyprus, Apr. 2005, pp. 280–286.

[55] J. Sachs, "A Generic Link Layer for Future Generation Wireless Networking," in *Proceedings of IEEE International Conference on Communications (ICC'03)*, Anchorage, AK, May 2003, pp. 834–838.

[56] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System Architecture Directions for Networked Sensors," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Nov. 2000, pp. 93–104.

[57] *Extensible Markup Language (XML) 1.0 Recomendation*, W3Consortium Std., February 1998. [Online]. Available: http://www.w3.org/TR/1998/REC-xml-19980210/

[58] P. Community, "Phython Programming Language," Aug. 2006, http://www.python.org/.

[59] T. Community, "Tcl Developer Xchange," Aug. 2006, http://www.tcl.tk/.

[60] "Simplified Wrapper and Interface Generator," Nov. 2006, http://www.swig.org/.

[61] S.-H. Choi, R. Grant, W. Kim, H. K. Wright, R. W. H. Jr., and S. M. Nettles, "An Experimental Evaluation of Several Rate Adaptation Protocols," in *Submitted to IEEE INFOCOM 08*, Phoenix, Arizona, Apr. 2008.

[62] K. Mandke, R. Daniels, S.-H. Choi, R. H. Jr., and S. Nettles, "Physical Concerns for Cross-layer Prototyping and Wireless Network experimentation," in *The Second ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, Montreal, Canada, Sept. 2007.

[63] V. Bharghavan, A. Demers, S. Shenker, and L. Zhang, "MACAW: A Media Access Protocol for Wireless LAN's," in *SIGCOMM*, 1994, pp. 212–225.

[64] F. Nait-Abdesselam, H. Koubaa, and W. Ding, "RAMAC: Routingaware Adaptive MAC in IEEE 802.11 Wireless Ad-Hoc Networks," in *Eighth International Conference on Cellular and Intelligent Communications (CIC'03)*, Seoul, Korea, Oct. 2003.

[65] M. Park, S.-H. Choi, and S. M. Nettles, "Cross-layer MAC Design for Wireless Networks using MIMO," in *Proceeding of the IEEE Global Telecommunications Conference (Globecom)*, St. Louise, MO, Nov. 2005.

[66] S.-H. Choi and S. M. Nettles, "An Overhead Controlled MAC Protocol for High Data-rate Wireless Networks," in *Proceedings of IEEE vehicular Technology Conference (VTC fall 05)*, Dallas, TX, Sept. 2005.

[67] H.-Y. Hsieh and R. Sivakumar, "IEEE 802.11 over Multi-hop Wireless Networks: Problems and New Perspectives," in *Proceedings of IEEE vehicular Technology Conference (VTC)*, Vancouver, Canada, Sept. 2002.

[68] H. Xia, Z. Zeng, and W. Ding, "Adaptive Backoff Algorithm Based on Network Congestion in Multi-hop Wireless Ad hoc Networks," in *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, Qingdao, China, July 2007.

[69] S. Xu and T. Saadawi, "Does the IEEE 802.11 MAC Protocol Work Well in Mul-

tihop Wireless Ad Hoc Networks?" *IEEE Communications Magazine*, vol. 39, pp. 130–137, June 2001.

[70] C. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers," in *Proceedings of the ACM SIGCOMM 1994 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, London, United Kingdom, Feb. 1994.

[71] D. Johnson and D. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks," *Mobile Computing*, vol. 353, 1996.

[72] C. Perkins and E. Royer, "Ad-hoc On-demand Distance Vector Routing," in *Proceedings of Mobile Computing Systems and Applications. (WMCSA '99)*, New Orleans, LA, Feb. 1999.

[73] T. Goff, N. Abu-Ghazaleh, D. Phatak, and R. Kahvecioglu, "Preemptive Routing in Ad Hoc Networks," in *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, Rome, Italy, July 2001.

[74] R. Punnoose, P. Nikitin, J. Broch, and D. Stancil, "Optimizing Wireless Network Protocols using Realtime Predictive Propagation Modeling," in *Proceedings of the IEEE Radio and Wireless Conference 1999 (RAWCON'99)*, Denver, CO, Aug. 1999.

[75] R. Dube, C. D. Rais, K.-Y. Wang, and S. K. Tripathi, "Signal Stability based Adaptive Routing (SSA) for Ad-Hoc Mobile Networks," *IEEE Personal Communication Magazine*, vol. 4, pp. 36–45, Feb. 1997.

[76] A. Adya, P. Bahl, J. Padhye, A. Wolman, and L. Zhou, "A Multi-Radio Unification Protocol for IEEE 802.11 Wireless Networks," in *Proceedings of the*

First International Conference on Broadband Networks (BROADNETS'04), San Jose, CA, Oct. 2004, pp. 344–354.

[77] B. Awerbuch, D. Holmer, and H. Rubens, "High Throughput Route Selection in Multi-rate Ad Hoc Wireless Networks," Technical report, Johns Hopkins University, Computer Science Department, Mar. 2003.

[78] S. Das, H. Pucha, K. Papagiannaki, and C. Hu, "Studying Wireless Routing Link Metric Dynamics," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, San Diego, CA, Oct. 2007, pp. 327–332.

[79] R. Draves, J. Padhye, and B. Zill, "Comparison of Routing Metrics for Static Multi-hop Wireless Networks," in *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Portland, OR, Aug. 2004, pp. 133 – 144.

[80] L. Peterson and B. Davie, *Computer Networks: A Systems Approach*. San Francisco, CA: Morgan kaufmamn, 2000.

Vita

Soon Hyeok Choi was born in Seoul, Korea on July 20, 1972. He received his Bachelor of Science and Master of Science degrees in Electrical Engineering from Yonsei University in February 1996 and February 1998, respectively. From 1998 to 2000, he worked for LG Electronics Inc. in Korea as a member of technical staff. From 2000 to 2002, he worked for C-EISA in Korea as a member of technical staff. In August 2002, he joined the Electrical and Computer Engineering at the University of Texas at Austin to pursue his doctoral degree. During his study, he was a member of the Wireless Networking and Communications Group.

Permanent Address: 103-1103 Seocho-samsung-lemian apartment
Seocho1-Dong Seocho-Gu
Seoul, 137-880, Korea

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.